Proceedings of the Linux Symposium

Volume One

June 27th–30th, 2007
Ottawa, Ontario
Canada

Contents

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

The Price of Safety: Evaluating IOMMU Performance

Muli Ben-Yehuda
IBM Haifa Research Lab
muli@il.ibm.com

Jimi Xenidis
IBM Research
jimix@watson.ibm.com

Michal Ostrowski
IBM Research
mostrows@watson.ibm.com

Karl Rister
IBM LTC
krister@us.ibm.com

Alexis Bruemmer
IBM LTC
alexisb@us.ibm.com

Leendert Van Doorn
AMD
Leendert.vanDoorn@amd.com

Abstract

IOMMUs, IO Memory Management Units, are hardware devices that translate device DMA addresses to machine addresses. An isolation capable IOMMU restricts a device so that it can only access parts of memory it has been explicitly granted access to. Isolation capable IOMMUs perform a valuable system service by preventing rogue devices from performing errant or malicious DMAs, thereby substantially increasing the system's reliability and availability. Without an IOMMU a peripheral device could be programmed to overwrite any part of the system's memory. Operating systems utilize IOMMUs to isolate device drivers; hypervisors utilize IOMMUs to grant secure direct hardware access to virtual machines. With the imminent publication of the PCI-SIG's IO Virtualization standard, as well as Intel and AMD's introduction of isolation capable IOMMUs in all new servers, IOMMUs will become ubiquitous.

Although they provide valuable services, IOMMUs can impose a performance penalty due to the extra memory accesses required to perform DMA operations. The exact performance degradation depends on the IOMMU design, its caching architecture, the way it is programmed and the workload. This paper presents the performance characteristics of the Calgary and DART IOMMUs in Linux, both on bare metal and in a hypervisor environment. The throughput and CPU utilization of several IO workloads, with and without an IOMMU, are measured and the results are analyzed. The potential strategies for mitigating the IOMMU's costs are then discussed. In conclusion a set of optimizations and resulting performance improvements are presented.

1 Introduction

An I/O Memory Management Unit (IOMMU) creates one or more unique address spaces which can be used to control how a DMA operation, initiated by a device, accesses host memory. This functionality was originally introduced to increase the addressability of a device or bus, particularly when 64-bit host CPUs were being introduced while most devices were designed to operate in a 32-bit world. The uses of IOMMUs were later extended to restrict the host memory pages that a device can actually access, thus providing an increased level of isolation, protecting the system from user-level device drivers and eventually virtual machines. Unfortunately, this additional logic does impose a performance penalty.

The wide spread introduction of IOMMUs by Intel [1] and AMD [2] and the proliferation of virtual machines will make IOMMUs a part of nearly every computer system. There is no doubt with regards to the benefits IOMMUs bring... but how much do they cost? We seek to quantify, analyze, and eventually overcome the performance penalties inherent in the introduction of this new technology.

1.1 IOMMU design

A broad description of current and future IOMMU hardware and software designs from various companies can be found in the OLS '06 paper entitled *Utilizing IOMMUs for Virtualization in Linux and Xen* [3]. The design of a system with an IOMMU can be broadly broken down into the following areas:

- IOMMU hardware architecture and design.
- Hardware $\leftrightarrow$ software interfaces.

- Pure software interfaces (e.g., between userspace and kernelspace or between kernelspace and hypervisor).

It should be noted that these areas can and do affect each other: the hardware/software interface can dictate some aspects of the pure software interfaces, and the hardware design dictates certain aspects of the hardware/software interfaces.

This paper focuses on two different implementations of the same IOMMU architecture that revolves around the basic concept of a Translation Control Entry (TCE). TCEs are described in detail in Section 1.1.2.

1.1.1 IOMMU hardware architecture and design

Just as a CPU-MMU requires a TLB with a very high hit-rate in order to not impose an undue burden on the system, so does an IOMMU require a translation cache to avoid excessive memory lookups. These translation caches are commonly referred to as IOTLBs.

The performance of the system is affected by several cache-related factors:

- The cache size and associativity [13].
- The cache replacement policy.
- The cache invalidation mechanism and the frequency and cost of invalidations.

The optimal cache replacement policy for an IOTLB is probably significantly different than for an MMU-TLB. MMU-TLBs rely on spatial and temporal locality to achieve a very high hit-rate. DMA addresses from devices, however, do not necessarily have temporal or spatial locality. Consider for example a NIC which DMAs received packets directly into application buffers: packets for many applications could arrive in any order and at any time, leading to DMAs to wildly disparate buffers. This is in sharp contrast with the way applications access their memory, where both spatial and temporal locality can be observed: memory accesses to nearby areas tend to occur closely together.

Cache invalidation can have an adverse effect on the performance of the system. For example, the *Calgary* IOMMU (which will be discussed later in detail) does not provide a software mechanism for invalidating a single cache entry—one must flush the entire cache to invalidate an entry. We present a related optimization in Section 4.

It should be mentioned that the PCI-SIG IOV (IO Virtualization) working group is working on an Address Translation Services (ATS) standard. ATS brings in another level of caching, by defining how I/O endpoints (i.e., adapters) inter-operate with the IOMMU to cache translations on the adapter and communicate invalidation requests from the IOMMU to the adapter. This adds another level of complexity to the system, which needs to be overcome in order to find the optimal caching strategy.

1.1.2 Hardware ↔ Software Interface

The main hardware/software interface in the TCE family of IOMMUs is the Translation Control Entry (TCE). TCEs are organized in TCE tables. TCE tables are analogous to page tables in an MMU, and TCEs are similar to page table entries (PTEs). Each TCE identifies a 4KB page of host memory and the access rights that the bus (or device) has to that page. The TCEs are arranged in a contiguous series of host memory pages that comprise the TCE table. The TCE table creates a single unique IO address space (DMA address space) for all the devices that share it.

The translation from a DMA address to a host memory address occurs by computing an index into the TCE table by simply extracting the page number from the DMA address. The index is used to compute a direct offset into the TCE table that results in a TCE that translates that IO page. The access control bits are then used to validate both the translation and the access rights to the host memory page. Finally, the translation is used by the bus to direct a DMA transaction to a specific location in host memory. This process is illustrated in Figure 1.

The TCE architecture can be customized in several ways, resulting in different implementations that are optimized for a specific machine. This paper examines the performance of two TCE implementations. The first one is the *Calgary* family of IOMMUs, which can be found in IBM's high-end *System x* (x86-64 based) servers, and the second one is the *DMA Address Relocation Table (DART)* IOMMU, which is often paired with PowerPC

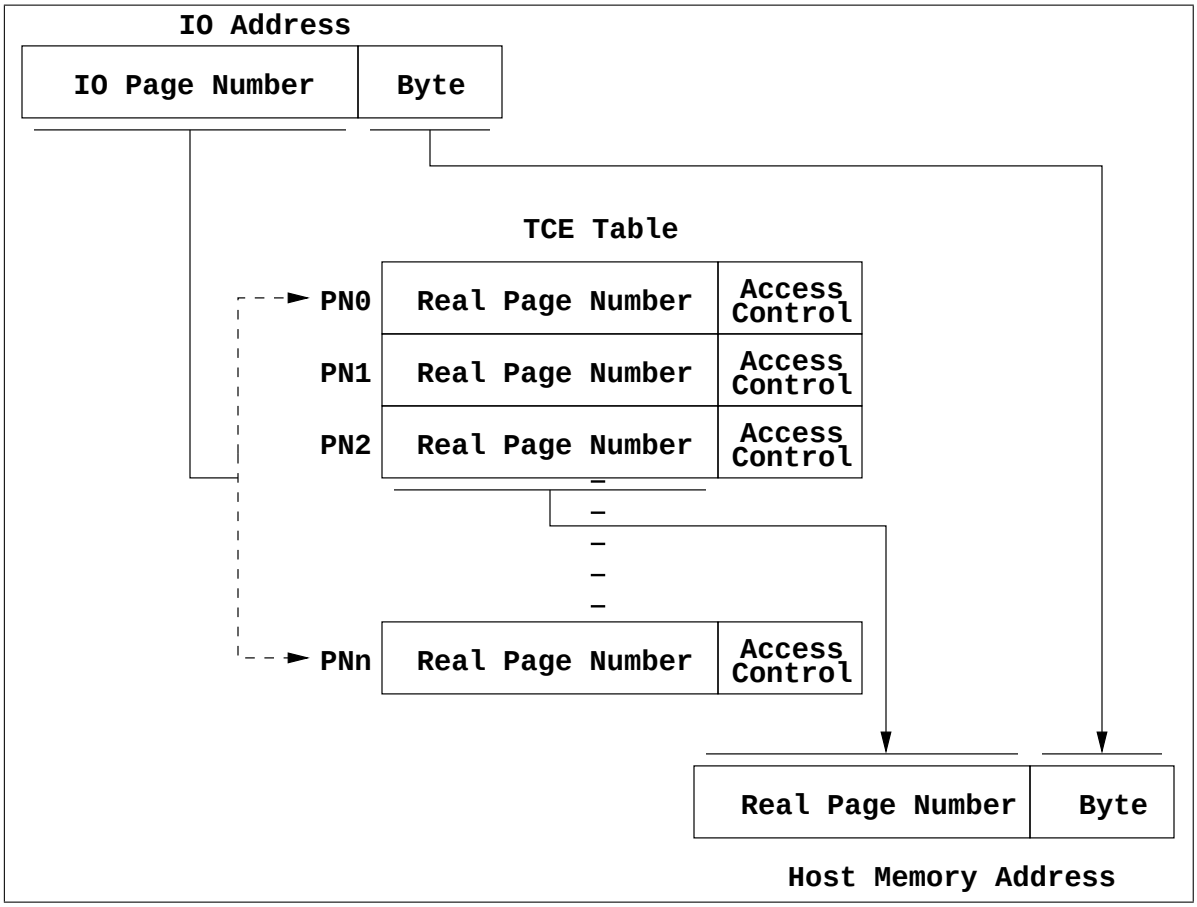


Figure 1: TCE table

970 processors that can be found in Apple G5 and IBM JS2x blades, as implemented by the CPC945 Bridge and Memory Controller.

The format of the TCEs are the first level of customization. Calgary is designed to be integrated with a Host Bridge Adapter or South Bridge that can be paired with several architectures—in particular ones with a huge addressable range. The Calgary TCE has the following format:

The 36 bits of RPN represent a generous 48 bits (256 TB) of addressability in host memory. On the other hand, the DART, which is integrated with the North Bridge of the Power970 system, can take advantage of the systems maximum 24-bit RPN for 36-bits (64 GB) of addressability and reduce the TCE size to 4 bytes, as shown in Table 2.

This allows DART to reduce the size of the table by half for the same size of IO address space, leading to better (smaller) host memory consumption and better host cache utilization.

Bits	Field	Description
0:15	Unused	
16:51	RPN	Real Page number
52:55	Reserved	
56:61	Hub ID	Used when a single TCE table isolates several busses
62	W*	W=1 ⇒ Write allowed
63	R*	R=1 ⇒ Read allowed
*R=0 and W=0 represent an invalid translation		

Table 1: Calgary TCE format

1.1.3 Pure Software Interfaces

The IOMMU is a shared hardware resource, which is used by drivers, which could be implemented in user-space, kernel-space, or hypervisor-mode. Hence the IOMMU needs to be owned, multiplexed and protected

Bits	Field	Description
0	Valid	1 - valid
1	R	R=0 ⇒ Read allowed
2	W	W=0 ⇒ Write allowed
3:7	Reserved	
8:31	RPN	Real Page Number

Table 2: DART TCE format

by system software—typically, an operating system or hypervisor.

In the bare-metal (no hypervisor) case, without any userspace driver, with Linux as the operating system, the relevant interface is Linux's DMA-API [4][5]. In-kernel drivers call into the DMA-API to establish and tear-down IOMMU mappings, and the IOMMU's DMA-API implementation maps and unmaps pages in the IOMMU's tables. Further details on this API and the Calgary implementation thereof are provided in the OLS '06 paper entitled *Utilizing IOMMUs for Virtualization in Linux and Xen* [3].

The hypervisor case is implemented similarly, with a hypervisor-aware IOMMU layer which makes hypercalls to establish and tear down IOMMU mappings. As will be discussed in Section 4, these basic schemes can be optimized in several ways.

It should be noted that for the hypervisor case there is also a common alternative implementation tailored for guest operating systems which are not aware of the IOMMU's existence, where the IOMMU's mappings are managed solely by the hypervisor without any involvement of the guest operating system. This mode of operation and its disadvantages are discussed in Section 4.3.1.

2 Performance Results and Analysis

This section presents the performance of IOMMUs, with and without a hypervisor. The benchmarks were run primarily using the *Calgary* IOMMU, although some benchmarks were also run with the *DART* IOMMU. The benchmarks used were FFSB [6] for disk IO and netperf [7] for network IO. Each benchmark was run in two sets of runs, first with the IOMMU disabled and then with the IOMMU enabled. The benchmarks were run on bare-metal Linux (Calgary and DART) and Xen dom0 and domU (Calgary).

For network tests the netperf [7] benchmark was used, using the TCP_STREAM unidirectional bulk data transfer option. The tests were run on an IBM x460 system (with the Hurricane 2.1 chipset), using 4 x dual-core Paxville Processors (with hyperthreading disabled). The system had 16GB RAM, but was limited to 4GB using mem=4G for IO testing. The system was booted and the tests were run from a QLogic 2300 Fiber Card (PCI-X, volumes from a DS3400 hooked to a SAN). The on-board Broadcom Gigabit Ethernet adapter was used. The system ran SLES10 x86_64 Base, with modified kernels and Xen.

The netperf client system was an IBM e326 system, with 2 x 1.8 GHz Opteron CPUs and 6GB RAM. The NIC used was the on-board Broadcom Gigabit Ethernet adapter, and the system ran an unmodified RHEL4 U4 distribution. The two systems were connected through a Cisco 3750 Gigabit Switch stack.

A 2.6.21-rc6 based tree with additional Calgary patches (which are expected to be merged for 2.6.23) was used for bare-metal testing. For Xen testing, the xen-iommu and linux-iommu trees [8] were used. These are IOMMU development trees which track xen-unstable closely. xen-iommu contains the hypervisor bits and linux-iommu contains the xenolinux (both dom0 and domU) bits.

2.1 Results

For the sake of brevity, we present only the network results. The FFSB (disk IO) results were comparable. For Calgary, the system was tested in the following modes:

- netperf server running on a bare-metal kernel.
- netperf server running in Xen dom0, with dom0 driving the IOMMU. This setup measures the performance of the IOMMU for a "direct hardware access" domain—a domain which controls a device for its own use.
- netperf server running in Xen domU, with dom0 driving the IOMMU and domU using virtual-IO (netfront or blkfront). This setup measures the performance of the IOMMU for a "driver domain" scenario, where a "driver domain" (dom0) controls a device on behalf of another domain (domU).

The first test (netperf server running on a bare-metal kernel) was run for DART as well.

Each set of tests was run twice, once with the IOMMU enabled and once with the IOMMU disabled. For each test, the following parameters were measured or calculated: throughput with the IOMMU disabled and enabled (off and on, respectively), CPU utilization with the IOMMU disabled and enabled, and the relative difference in throughput and CPU utilization. Note that due to different setups the CPU utilization numbers are different between bare-metal and Xen. Each CPU utilization number is accompanied by the potential maximum.

For the bare-metal network tests, summarized in Figures 2 and 3, there is practically no difference between the CPU throughput with and without an IOMMU. With an IOMMU, however, the CPU utilization can be as much as 60% more (!), albeit it is usually closer to 30%. These results are for Calgary—for DART, the results are largely the same.

For Xen, tests were run with the netperf server in dom0 as well as in domU. In both cases, dom0 was driving the IOMMU (in the tests where the IOMMU was enabled). In the domU tests domU was using the virtual-IO drivers. The dom0 tests measure the performance of the IOMMU for a "direct hardware access" scenario and the domU tests measure the performance of the IOMMU for a "driver domain" scenario.

Network results for netperf server running in dom0 are summarized in Figures 4 and 5. For messages of sizes 1024 and up, the results strongly resemble the bare-metal case: no noticeable throughput difference except for very small packets and 40–60% more CPU utilization when IOMMU is enabled. For messages with sizes of less than 1024, the throughput is significantly less with the IOMMU enabled than it is with the IOMMU disabled.

For Xen domU, the tests show up to 15% difference in throughput for message sizes smaller than 512 and up to 40% more CPU utilization for larger messages. These results are summarized in Figures 6 and 7.

3 Analysis

The results presented above tell mostly the same story: throughput is the same, but CPU utilization rises when the IOMMU is enabled, leading to up to 60% more CPU utilization. The throughput difference with small network message sizes in the Xen network tests probably stems from the fact that the CPU isn't able to keep up with the network load when the IOMMU is enabled. In other words, dom0's CPU is close to the maximum even with the IOMMU disabled, and enabling the IOMMU pushes it over the edge.

On one hand, these results are discouraging: enabling the IOMMU to get safety and paying up to 60% more in CPU utilization isn't an encouraging prospect. On the other hand, the fact that the throughput is roughly the same when the IOMMU code doesn't overload the system strongly suggests that software is the culprit, rather than hardware. This is good, because software is easy to fix!

Profile results from these tests strongly suggest that mapping and unmapping an entry in the TCE table is the biggest performance hog, possibly due to lock contention on the IOMMU data structures lock. For the bare-metal case this operation does not cross address spaces, but it does require taking a spinlock, searching a bitmap, modifying it, performing several arithmetic operations, and returning to the user. For the hypervisor case, these operations require all of the above, *as well as switching to hypervisor mode*.

As we will see in the next section, most of the optimizations discussed are aimed at reducing both the number and costs of TCE map and unmap requests.

4 Optimizations

This section discusses a set of optimizations that have either already been implemented or are in the process of being implemented. "Deferred Cache Flush" and "Xen multicalls" were implemented during the IOMMU's bring-up phase and are included in the results presented above. The rest of the optimizations are being implemented and were not included in the benchmarks presented above.

4.1 Deferred Cache Flush

The Calgary IOMMU, as it is used in Intel-based servers, does not include software facilities to invalidate selected entries in the TCE cache (IOTLB). The only

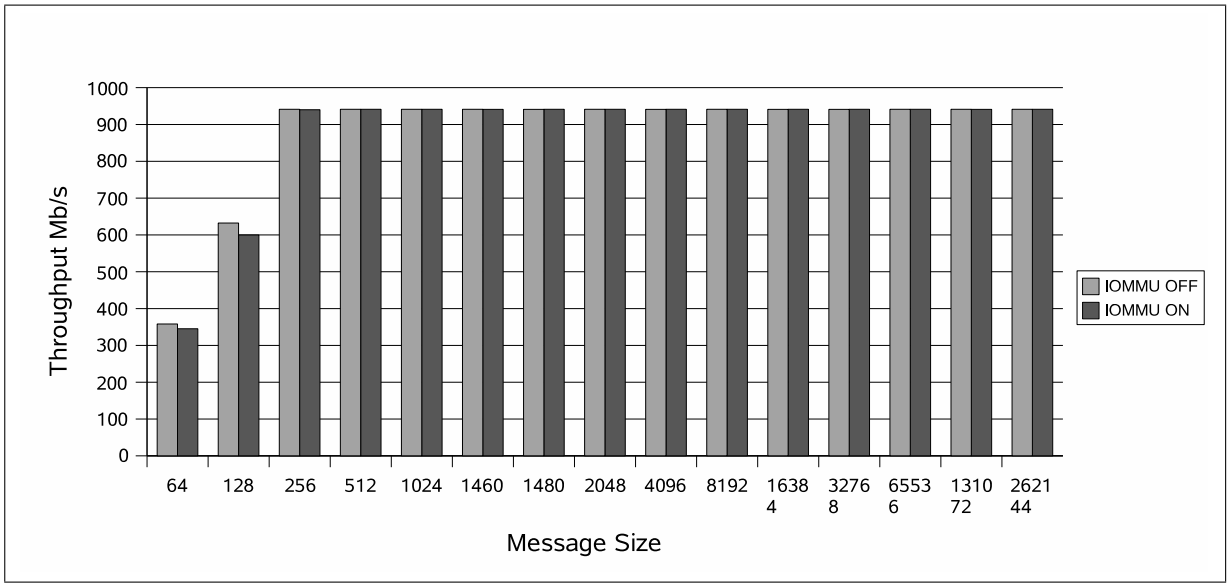


Figure 2: Bare-metal Network Throughput

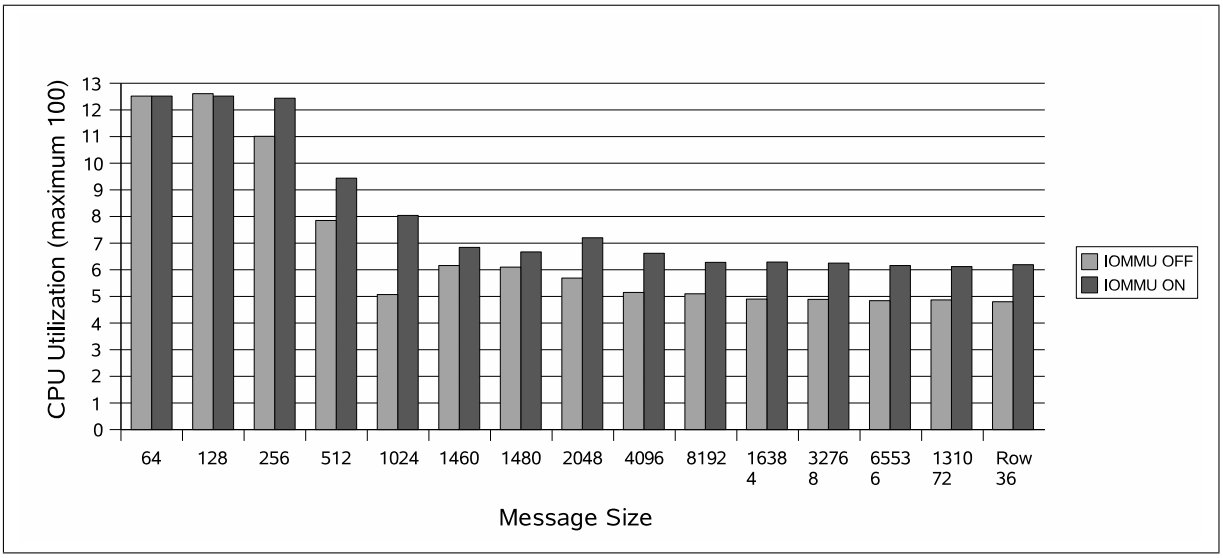


Figure 3: Bare-metal Network CPU Utilization

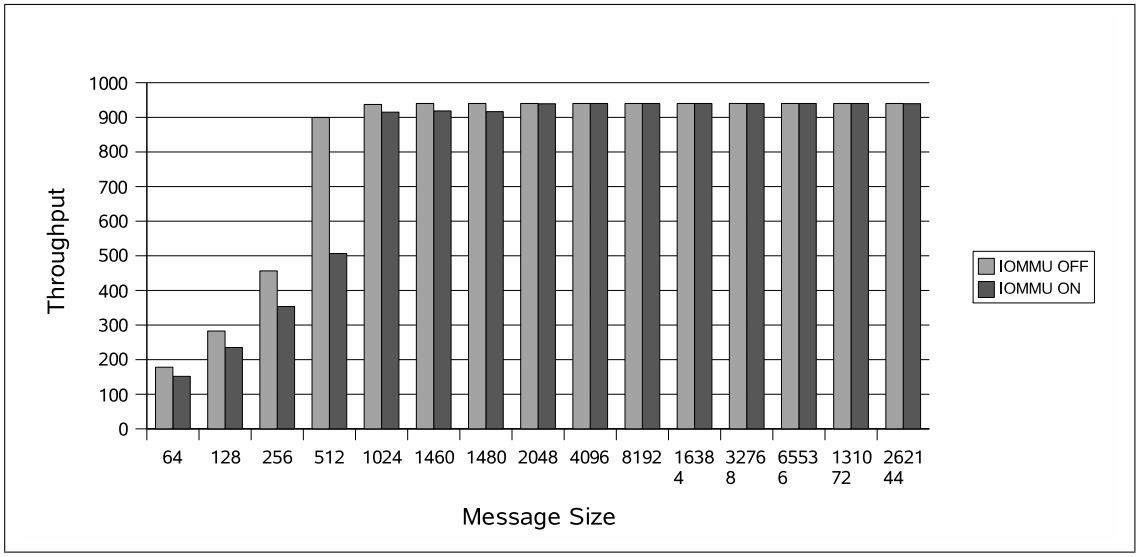


Figure 4: Xen dom0 Network Throughput

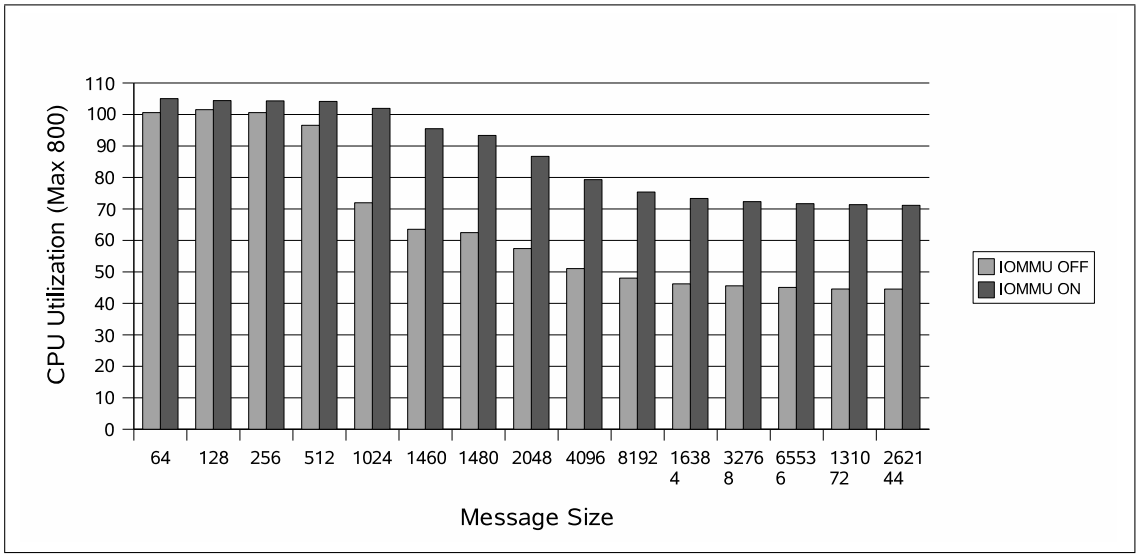


Figure 5: Xen dom0 Network CPU Utilization

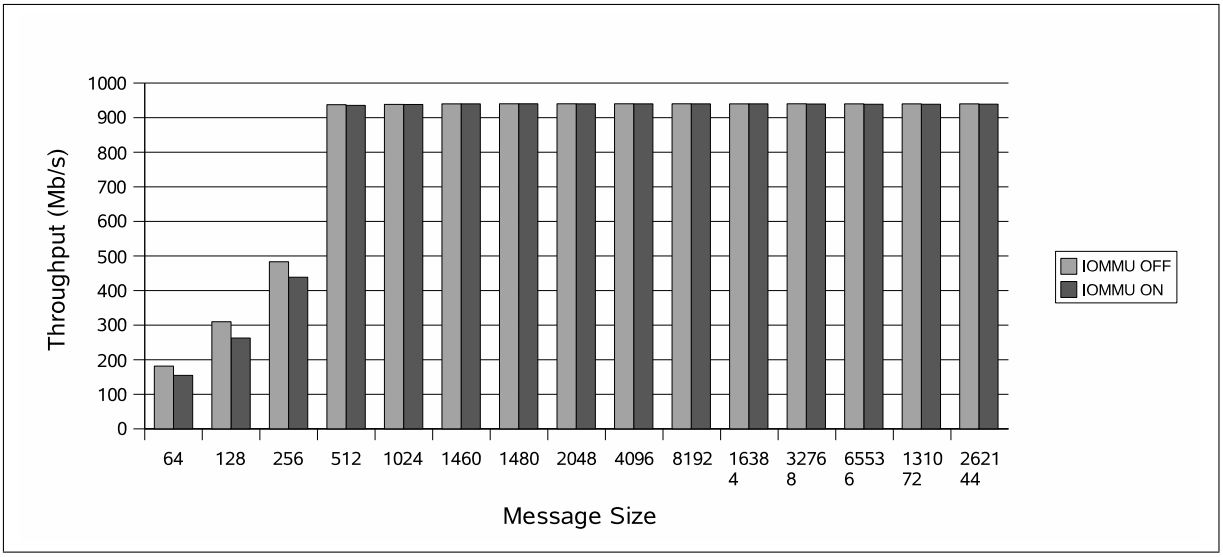


Figure 6: Xen domU Network Throughput

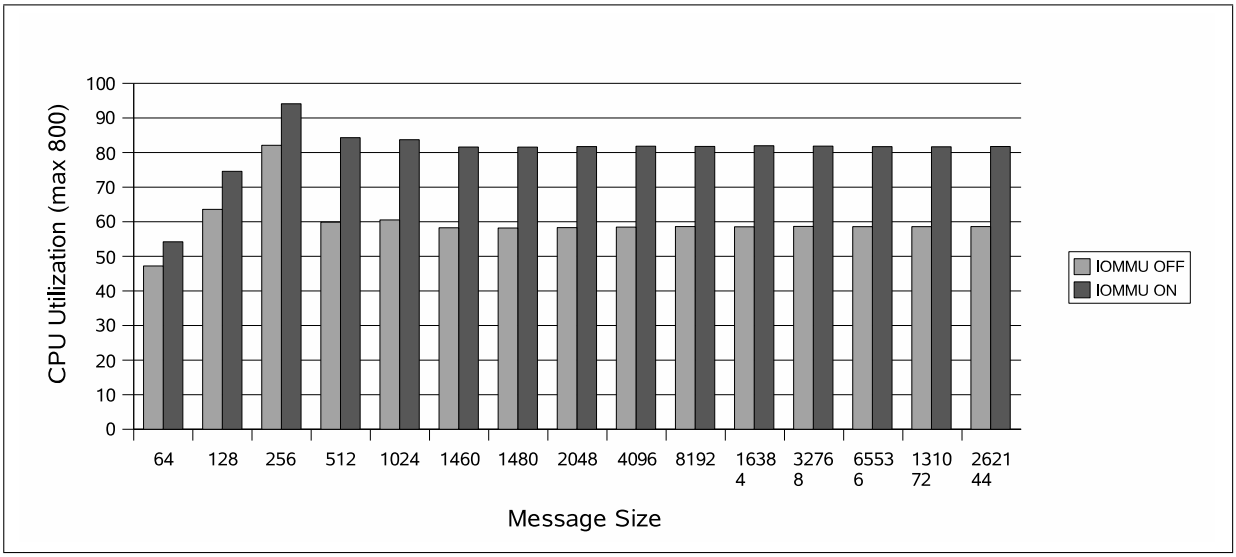


Figure 7: Xen domU Network CPU Utilization

way to invalidate an entry in the TCE cache is to quiesce all DMA activity in the system, wait until all outstanding DMAs are done, and then flush the entire TCE cache. This is a cumbersome and lengthy procedure.

In theory, for maximal safety, one would want to invalidate an entry as soon as that entry is unmapped by the driver. This will allow the system to catch any "use after free" errors. However, flushing the entire cache after every unmap operation proved prohibitive—it brought the system to its knees. Instead, the implementation trades a little bit of safety for a whole lot of usability. Entries in the TCE table are allocated using a next-fit allocator, and the cache is only flushed when the allocator rolls around (starts to allocate from the beginning). This optimization is based on the observation that an entry only needs to be invalidated before it is re-used. Since a given entry will only be reused once the allocator rolls around, roll-around is the point where the cache must be flushed.

The downside to this optimization is that it is possible for a driver to reuse an entry after it has unmapped it, *if* that entry happened to remain in the TCE cache. Unfortunately, closing this hole by invalidating every entry immediately when it is freed, cannot be done with the current generation of the hardware. The hole has never been observed to occur in practice.

This optimization is applicable to both bare-metal and hypervisor scenarios.

4.2 Xen multicalls

The Xen hypervisor supports "multicalls" [12]. A multicall is a single hypercall that includes the parameters of several distinct logical hypercalls. Using multicalls it is possible to reduce the number of hypercalls needed to perform a sequence of operations, thereby reducing the number of address space crossings, which are fairly expensive.

The Calgary Xen implementation uses multicalls to communicate map and unmap requests from a domain to the hypervisor. Unfortunately, profiling has shown that the vast majority of map and unmap requests (over 99%) are for a single entry, making multicalls pointless.

This optimization is only applicable to hypervisor scenarios.

4.3 Overhauling the DMA API

Profiling of the above mentioned benchmarks shows that the number one culprits for CPU utilization are the map and unmap calls. There are several ways to cut down on the overhead of map and unmap calls:

- Get rid of them completely.
- Allocate in advance; free when done.
- Allocate and free in large batches.
- Never free.

4.3.1 Using Pre-allocation to Get Rid of Map and Unmap

Calgary provides somewhat less than a 4GB DMA address space (exactly how much less depends on the system's configuration). If the guest's pseudo-physical address space fits within the DMA address space, one possible optimization is to only allocate TCEs when the guest starts up and free them when the guest shuts down. The TCEs are allocated such that TCE i maps the same machine frame as the guest's pseudo-physical address i. Then the guest could pretend that it doesn't have an IOMMU and pass the pseudo-physical address directly to the device. No cache flushes are necessary because no entry is ever invalidated.

This optimization, while appealing, has several downsides: first and foremost, it is only applicable to a hypervisor scenario. In a bare-metal scenario, getting rid of map and unmap isn't practical because it renders the IOMMU useless—if one maps all of physical memory, why use an IOMMU at all? Second, even in a hypervisor scenario, pre-allocation is only viable if the set of machine frames owned by the guest is "mostly constant" through the guest's lifetime. If the guest wishes to use page flipping or ballooning, or any other operation which modifies the guest's pseudo-physical to machine mapping, the IOMMU mapping needs to be updated as well so that the IO to machine mapping will again correspond exactly to the pseudo-physical to machine mapping. Another downside of this optimization is that it protects other guests and the hypervisor from the guest, but provides no protection inside the guest itself.

4.3.2 Allocate In Advance And Free When Done

This optimization is fairly simple: rather than using the "streaming" DMA API operations, use the alloc and free operations to allocate and free DMA buffers and then use them for as long as possible. Unfortunately this requires a massive change to the Linux kernel since driver writers have been taught since the days of yore that DMA mappings are a sparse resource and should only be allocated when absolutely needed. A better way to do this might be to add a caching layer inside the DMA API for platforms with many DMA mappings so that driver writers could still use the map and unmap API, but the actual mapping and unmapping will only take place the first time a frame is mapped. This optimization is applicable to both bare-metal and hypervisors.

4.3.3 Allocate And Free In Large Batches

This optimization is a twist on the previous one: rather than modifying drivers to use alloc and free rather than map and unmap, use map_multi and unmap_multi wherever possible to batch the map and unmap operations. Again, this optimization requires fairly large changes to the drivers and subsystems and is applicable to both bare-metal and hypervisor scenarios.

4.3.4 Never Free

One could sacrifice some of the protection afforded by the IOMMU for the sake of performance by simply never unmapping entries from the TCE table. This will reduce the cost of unmap operations (but not eliminate it completely—one would still need to know which entries are mapped and which have been theoretically "unmapped" and could be reused) and will have a particularly large effect on the performance of hypervisor scenarios. However, it will sacrifice a large portion of the IOMMU's advantage: any errant DMA to an address that corresponds with a previously mapped and unmapped entry will go through, causing memory corruption.

4.4 Grant Table Integration

This work has mostly been concerned with "direct hardware access" domains which have direct access to hardware devices. A subset of such domains are Xen "driver domains" [11], which use direct hardware access to perform IO on behalf of other domains. For such "driver domains," using Xen's grant table interface to pre-map TCE entries as part of the grant operation will save an address space crossing to map the TCE through the DMA API later. This optimization is only applicable to hypervisor (specifically, Xen) scenarios.

5 Future Work

Avenues for future exploration include support and performance evaluation for more IOMMUs such as Intel's VT-d [1] and AMD's IOMMU [2], completing the implementations of the various optimizations that have been presented in this paper and studying their effects on performance, coming up with other optimizations and ultimately gaining a better understanding of how to build "zero-cost" IOMMUs.

6 Conclusions

The performance of two IOMMUs, DART on PowerPC and Calgary on x86-64, was presented, through running IO-intensive benchmarks with and without an IOMMU on the IO path. In the common case throughput remained the same whether the IOMMU was enabled or disabled. CPU utilization, however, could be as much as 60% more in a hypervisor environment and 30% more in a bare-metal environment, when the IOMMU was enabled.

The main CPU utilization cost came from too-frequent map and unmap calls (used to create translation entries in the DMA address space). Several optimizations were presented to mitigate that cost, mostly by batching map and unmap calls in different levels or getting rid of them entirely where possible. Analyzing the feasibility of each optimization and the savings it produces is a work in progress.

Acknowledgments

The authors would like to thank Jose Renato Santos and Yoshio Turner for their illuminating comments and questions on an earlier draft of this manuscript.

References

[1] *Intel Virtualization Technology for Directed I/O Architecture Specification*, 2006, ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf.

[2] *AMD I/O Virtualization Technology (IOMMU)* Specification, 2006, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf.

[3] *Utilizing IOMMUs for Virtualization in Linux and Xen*, by M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. Van Doorn, A. Mallick, J. Nakajima, and E. Wahlig, in Proceedings of the 2006 Ottawa Linux Symposium (OLS), 2006.

[4] *Documentation/DMA-API.txt*.

[5] *Documentation/DMA-mapping.txt*.

[6] *Flexible Filesystem Benchmark (FFSB)* http://sourceforge.net/projects/ffsb/

[7] *Netperf Benchmark* http://www.netperf.org

[8] *Xen IOMMU trees*, 2007, http://xenbits.xensource.com/ext/xen-iommu.hg, http://xenbits.xensource.com/ext/linux-iommu.hg

[9] *Xen and the Art of Virtualization*, by B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, in Proceedings of the 19th ASM Symposium on Operating Systems Principles (SOSP), 2003.

[10] *Xen 3.0 and the Art of Virtualization*, by I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, in Proceedings of the 2005 Ottawa Linux Symposium (OLS), 2005.

[11] *Safe Hardware Access with the Xen Virtual Machine Monitor*, by K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, M. Williamson, in Proceedings of the OASIS ASPLOS 2004 workshop, 2004.

[12] *Virtualization in Xen 3.0*, by R. Rosen, http://www.linuxjournal.com/article/8909

[13] *Computer Architecture, Fourth Edition: A Quantitative Approach*, by J. Hennessy and D. Patterson, Morgan Kaufmann publishers, September, 2006.

Linux on Cell Broadband Engine status update

Arnd Bergmann
IBM Linux Technology Center
arnd.bergmann@de.ibm.com

Abstract

With Linux for the Sony PS3, the IBM QS2x blades and the Toshiba Celleb platform having hit mainstream Linux distributions, programming for the Cell BE is becoming increasingly interesting for developers of performance computing. This talk is about the concepts of the architecture and how to develop applications for it.

Most importantly, there will be an overview of new feature additions and latest developments, including:

- Preemptive scheduling on SPUs (finally!): While it has been possible to run concurrent SPU programs for some time, there was only a very limited version of the scheduler implemented. Now we have a full time-slicing scheduler with normal and real-time priorities, SPU affinity and gang scheduling.
- Using SPUs for offloading kernel tasks: There are a few compute intensive tasks like RAID-6 or IPsec processing that can benefit from running partially on an SPU. Interesting aspects of the implementation are how to balance kernel SPU threads against user processing, how to efficiently communicate with the SPU from the kernel and measurements to see if it is actually worthwhile.
- Overlay programming: One significant limitation of the SPU is the size of the local memory that is used for both its code and data. Recent compilers support overlays of code segments, a technique widely known in the previous century but mostly forgotten in Linux programming nowadays.

1 Background

The architecture of the Cell Broadband Engine (Cell/B.E.) is unique in many ways. It combines a general purpose PowerPC processor with eight highly optimized vector processing cores called the Synergistic Processing Elements (SPEs) on a single chip. Despite implementing two distinct instruction sets, they share the design of their memory management units and can access virtual memory in a cache-coherent way.

The Linux operating system runs on the PowerPC Processing Element (PPE) only, not on the SPEs, but the kernel and associated libraries allow users to run special-purpose applications on the SPE as well, which can interact with other applications running on the PPE. This approach makes it possible to take advantage of the wide range of applications available for Linux, while at the same time utilize the performance gain provided by the SPE design, which could not be achieved by just recompiling regular applications for a new architecture.

One key aspect of the SPE design is the way that memory access works. Instead of a cache memory that speeds up memory accesses in most current designs, data is always transferred explicitly between the local on-chip SRAM and the virtually addressed system memory. An SPE program resides in the local 256KiB of memory, together with the data it is working on. Every time it wants to work on some other data, the SPE tells its Memory Flow Controller (MFC) to asynchronously copy between the local memory and the virtual address space.

The advantage of this approach is that a well-written application practically never needs to wait for a memory access but can do all of these in the background. The disadvantages include the limitation to 256KiB of directly addressable memory that limit the set of applications that can be ported to the architecture, and the relatively long time required for a context switch, which needs to save and restore all of the local memory and the state of ongoing memory transfers instead of just the CPU registers.

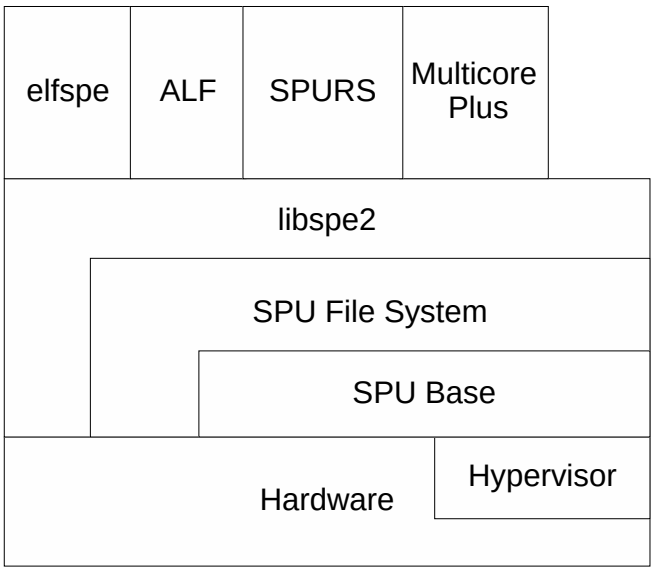


Figure 1: Stack of APIs for accessing SPEs

1.1 Linux port

Linux on PowerPC has a concept of platform types that the kernel gets compiled for, there are for example separate platforms for IBM System p and the Apple Power Macintosh series. Each platform has its own hardware specific code, but it is possible to enable combinations of platforms simultaneously. For the Cell/B.E., we initially added a platform named "cell" to the kernel, which has the drivers for running on the bare metal, i.e. without a hypervisor. Later, the code for both the Toshiba Celleb platform and Sony's PlayStation 3 platform were added, because each of them have their own hypervisor abstractions that are incompatible with each other and with the hypervisor implementations from IBM. Most of the code that operates on SPEs however is shared and provides a common interface to user processes.

2 Programming interfaces

There is a variety of APIs available for using SPEs, I'll try to give an overview of what we have and what they are used for. For historic reasons, the kernel and toolchain refer to SPUs (Synergistic Processing Units) instead of SPEs, of which they are strictly speaking a subset. For practical purposes, these two terms can be considered equivalent.

2.1 Kernel SPU base

There is a common interface for simple users of an SPE in the kernel, the main purpose is to make it possible to implement the SPU file system (spufs). The SPU base takes care of probing for available SPEs in the system and mapping their registers into the kernel address space. The interface is provided by the `include/asm-powerpc/spu.h` file. Some of the registers are only accessible through hypervisor calls on platforms where Linux runs virtualized, so accesses to these registers get abstracted by indirect function calls in the base.

A module that wants to use the SPU base needs to request a handle to a physical SPU and provide interrupt handler callbacks that will be called in case of events like page faults, stop events or error conditions.

The SPU file system is currently the only user of the SPU base in the kernel, but some people have implemented experimental other users, e.g. for acceleration of device drivers with SPUs inside of the kernel. Doing this is an easy way for prototyping kernel code, but we are recommending the use of spufs even from inside the kernel for code that you intend to have merged upstream. Note that as in-kernel interfaces, the API of the SPU base is not stable and can change at any time. All of its symbols are exported only to GPL-licensed users.

2.2 The SPU file system

The SPU file system provides the user interface for accessing SPUs from the kernel. Similar to procfs and sysfs, it is a purely virtual file system and has no block device as its backing. By convention, it gets mounted world-writable to the `/spu` directory in the root file system.

Directories in spufs represent SPU contexts, whose properties are shown as regular files in them. Any interaction with these contexts is done through file operation like read, write or mmap. At time of this writing, there are 30 files that are present in the directory of an SPU context, I will describe some of them as an example later.

Two system calls have been introduced for use exclusively together with spufs, spu_create and spu_run. The spu_create system call creates an SPU context in the kernel and returns an open file descriptor for the directory

associated with it. The open file descriptor is significant, because it is used as a measure to determine the life time of the context, which is destroyed when the file descriptor is closed.

Note the explicit difference between an SPU context and a physical SPU. An SPU context has all the properties of an actual SPU, but it may not be associated with one and only exists in kernel memory. Similar to task switching, SPU contexts get loaded into SPUs and removed from them again by the kernel, and the number of SPU contexts can be larger than the number of available SPUs.

The second system call, spu_run, acts as a switch for a Linux thread to transfer the flow of control from the PPE to the SPE. As seen by the PPE, a thread calling spu_run blocks in that system call for an indefinite amount of time, during which the SPU context is loaded into an SPU and executed there. An equivalent to spu_run on the SPU itself is the stop-and-signal instruction, which transfers control back to the PPE. Since an SPE does not run signal handlers itself, any action on the SPE that triggers a signal or others sending a signal to the thread also cause it to stop on the SPE and resume running on the PPE.

Files in a context include

mem The mem file represents the local memory of an SPU context. It can be accessed as a linear file using read/write/seek or mmap operation. It is fully transparent to the user whether the context is loaded into an SPU or saved to kernel memory, and the memory map gets redirected to the right location on a context switch. The most important use of this file is for an object file to get loaded into an SPU before it is run, but mem is also used frequently by applications themselves.

regs The general purpose registers of an SPU can not normally be accessed directly, but they can be in a saved context in kernel memory. This file contains a binary representation of the registers as an array of 128-bit vector variables. While it is possible to use read/write operations on the regs file in order to set up a newly loaded program or for debugging purposes, every access to it means that the context gets saved into a kernel save area, which is an expensive operation.

wbox The wbox file represents one of three mail box files that can be used for unidirectional communication between a PPE thread and a thread running on the SPE. Similar to a FIFO, you can not seek in this file, but only write data to it, which can be read using a special blocking instruction on the SPE.

phys-id The phys-id does not represent a feature of a physical SPU but rather presents an interface to get auxiliary information from the kernel, in this case the number of the SPU that a context is loaded into, or -1 if it happens not to be loaded at all at the point it is read. We will probably add more files with statistical information similar to this one, to give users better analytical functions, e.g. with an implementation of top that knows about SPU utilization.

2.3 System call vs. direct register access

Many functions of spufs can be accessed through two different ways. As described above, there are files representing the registers of a physical SPU for each context in spufs. Some of these files also allow the mmap() operation that puts a register area into the address space of a process.

Accessing the registers from user space through mmap can significantly reduce the system call overhead for frequent accesses, but it carries a number of disadvantages that users need to worry about:

- When a thread attempts to read or write a register of an SPU context running in another thread, a page fault may need to be handled by the kernel. If that context has been moved to the context save area, e.g. as the result of preemptive scheduling, the faulting thread will not make any progress until the SPU context becomes running again. In this case, direct access is significantly slower than indirect access through file operations that are able to modify the saved state.
- When a thread tries to access its own registers while it gets unloaded, it may block indefinitely and need to be killed from the outside.
- Not all of the files that can get mapped on one kernel version can be on another one. When using 64k pages, some files can not be mapped due to hardware restrictions, and some hypervisor implementations put different limitation on what can be mapped. This makes it very hard to write portable applications using direct mapping.

- In concurrent access to the registers, e.g. two threads writing simultaneously to the mailbox, the user application needs to provide its own locking mechanisms, as the kernel can not guarantee atomic accesses.

In general, application writers should use a library like libspe2 to do the abstraction. This library contains functions to access the registers with correct locking and provides a flag that can be set to attempt using the direct mapping or fall back to using the safe file system access.

2.4 elfspe

For users that want to worry as little as possible about the low-level interfaces of spufs, the elfspe helper is the easiest solution. Elfspe is a program that takes an SPU ELF executable and loads it into a newly created SPU context in spufs. It is able to handle standard callbacks from a C library on the SPU, which are needed e.g. to implement printf on the SPU by running some of code on the PPE.

By installing elfspe with the miscellaneous binary format kernel support, the kernel execve() implementation will know about SPU executables and use `/sbin/elfspe` as the interpreter for them, just like it calls interpreters for scripts that start with the well-known "#!" sequence.

Many programs that use only the subset of library functions provided by newlib, which is a C runtime library for embedded systems, and fit into the limited local memory of an SPE are instantly portable using elfspe. Important functionalities that does not work with this approach include:

shared libraries Any library that the executable needs also has to be compiled for the SPE and its size adds up to what needs to fit into the local memory. All libraries are statically linked.

threads An application using elfspe is inherently single-threaded. It can neither use multiple SPEs nor multiple threads on one SPE.

IPC Inter-process communication is significantly limited by what is provided through newlib. Use of system calls directly from an SPE is not easily available with the current version of elfspe, and any interface that requires shared memory requires special adaptation to the SPU environment in order to do explicit DMA.

2.5 libspe2

Libspe2 is an implementation of the operating-system-independent "SPE Runtime Management Library" specification.[1] This is what most applications are supposed to be written for in order to get the best degree of portability. There was an earlier libspe 1.x, that is not actively maintained anymore since the release of version 2.1.

Unlike elfspe, libspe2 requires users to maintain SPU contexts in their own code, but it provides an abstraction from the low-level spufs details like file operations, system calls and register access.

Typically, users want to have access to more than one SPE from one application, which is typically done through multithreading the program: each SPU context gets its own thread that calls the spu_run system call through libspe2. Often, there are additional threads that do other work on the PPE, like communicating with the running SPE threads or providing a GUI. In a program where the PPE hands out tasks to the SPEs, libspe2 provides event handles that the user can call blocking functions like `epoll_wait()` on to wait for SPEs requesting new data.

2.6 Middleware

There are multiple projects targeted at providing a layer on top of libspe2 to add application-side scheduling of jobs inside of an SPU context. These include the SPU Runtime System (SPURS) from Sony, the Accelerator Library Framework (ALF) from IBM and the MultiCore Plus SDK from Mercury Computer Systems.

All these projects have in common that there is no public documentation or source code available at this time, but that will probably change in the time until the Linux Symposium.

[1] `http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/1DFEF31B321111258725724200 7883F3/$file/cplibspe.pdf`

3 SPU scheduling

While spufs has had the concept of abstracting SPU contexts from physical SPUs from the start, there has not been any proper scheduling for a long time. An initial implementation of a preemptive scheduler was first merged in early 2006, but then disabled again as there were too many problems with it.

After a lot of discussion, a new implementation of the SPU scheduler from Christoph Hellwig has been merged in the 2.6.20 kernel, initially only supporting only SCHED_RR and SCHED_FIFO real-time priority tasks to preempt other tasks, but later work was done to add time slicing as well for regular SCHED_OTHER threads.

Since SPU contexts do not directly correspond to Linux threads, the scheduler is independent of the Linux process scheduler. The most important difference is that a context switch is performed by the kernel, running on the PPE, not by the SPE, which the context is running on.

The biggest complication when adding the scheduler is that a number of interfaces expect a context to be in a specific state. Accessing the general purpose registers from GDB requires the context to be saved, while accessing the signal notification registers through mmap requires the context to be running. The new scheduler implementation is conceptually simpler than the first attempt in that no longer attempts to schedule in a context when it gets accessed by someone else, but rather waits for the context to be run by means of another thread calling spu_run.

Accessing one SPE from another one shows effects of non-uniform memory access (NUMA) and application writers typically want to keep a high locality between threads running on different SPEs and the memory they are accessing. The SPU code therefore has been able for some time to honor node affinity settings done through the NUMA API. When a thread is bound to a given CPU while executing on the PPE, spufs will implicitly bind the thread to an SPE on the same physical socket, to the degree that relationship is described by the firmware.

This behavior has been kept with the new scheduler, but has been extended by another aspect, affinity between SPE cores on the same socket. Unlike the NUMA interfaces, we don't bind to a specific core here, but describe the relationship between SPU contexts. The spu_create system call now gets an optional argument that lets the user pass the file descriptor of an existing context. The spufs scheduler will then attempt to move these contexts to physical SPEs that are close on the chip and can communicate with lower overhead than distant ones.

Another related interface is the temporal affinity between threads. If the two threads that you want to communicate with each other don't run at the same time, the special affinity is pointless. A concept called gang scheduling is applied here, with a gang being a container of SPU contexts that are all loaded simultaneously. A gang is created in spufs by passing a special flag to spu_create, which then returns a descriptor to an empty gang directory. All SPU contexts created inside of that gang are guaranteed to be loaded at the same time.

In order to limit the number of expensive operations of context switching an entire gang, we apply lazy context switching to the contexts in a gang. This means we don't load any contexts into SPUs until all contexts in the gang are waiting in spu_run to become running. Similarly, when one of the threads stops, e.g. because of a page fault, we don't immediately unload the contexts but wait until the end of the time slice. Also, like normal (non-gang) contexts, the gang will not be removed from the SPUs unless there is actually another thread waiting for them to become available, independent of whether or not any of the threads in the gang execute code at the end of the time slice.

4 Using SPEs from the kernel

As mentioned earlier, the SPU base code in the kernel allows any code to get access to SPE resources. However, that interface has the disadvantage to remove the SPE from the scheduling, so valuable processing power remains unused while the kernel is not using the SPE. That should be most of the time, since compute-intensive tasks should not be done in kernel space if possible.

For tasks like IPsec, RAID6 or dmcrypt processing offload, we usually want the SPE to be only blocked while the disk or network is actually being accessed, otherwise it should be available to user space.

Sebastian Siewior is working on code to make it possible to use the spufs scheduler from the kernel, with the concrete goal of providing cryptoapi offload functions for common algorithms.

For this, the in-kernel equivalent of libspe is created, with functions that directly do low-level accesses instead of going through the file system layer. Still, the SPU contexts are visible to user space applications, so they can get statistic information about the kernel space SPUs.

Most likely, there should be one kernel thread per SPU context used by the kernel. It should also be possible to have multiple unrelated functions that are offloaded from the kernel in the same executable, so that when the kernel needs one of them, it calls into the correct location on the SPU. This requires some infrastructure to link the SPU objects correctly into a single binary. Since the kernel does not know about the SPU ELF file format, we also need a new way of initially loading the program into the SPU, e.g. by creating a save context image as part of the kernel build process.

First experiments suggest that an SPE can do an AES encryption about four times faster than a PPE. It will need more work to see if that number can be improved further, and how much of it is lost as communication overhead when the SPE needs to synchronize with the kernel. Another open question is whether it is more efficient for the kernel to synchronously wait for the SPE or if it can do something else at the same time.

5 SPE overlays

One significant limitation of the SPE is the size that is available for object code in the local memory. To overcome that limitation, new binutils support overlay to support overlaying ELF segments into concurrent regions. In the most simple case, you can have two functions that both have their own segment, with the two segments occupying the same region. The size of the region is the maximum of either segment size, since they both need to fit in the same space.

When a function in an overlay is called, the calling function first needs to call a stub that checks if the correct overlay is currently loaded. If not, a DMA transfer is initiated that loads the new overlay segment, overwriting the segment loaded into the overlay region before. This makes it possible to even do function calls in different segments of the same region.

There can be any number of segments per region, and the number of regions is only limited by the size of the local storage. However, the task of choosing the optimal configuration of which functions to go into what segment is up to the application developer. It gets specified through a linker script that contains a list of OVERLAY statements, each of them containing a list of segments that go into an overlay.

It is only possible to overlay code and read-only data, but not data that is written to, because overlay segments only ever get loaded into the SPU, but never written back to main memory.

6 Profiling SPE tasks

Support for profiling SPE tasks with the oprofile tool has been implemented in the latest IBM Software Development Kit for Cell. It is currently in the process of getting merged into the mainline kernel and oprofile user space packages.

It uses the debug facilities provided by the Cell/B.E. hardware to get sample data about what each SPE is doing, and then maps that to currently running SPU contexts. When the oprofile report tool runs, that data can be mapped back to object files and finally to source code lines that a developer can understand. So far, it behaves like oprofile does for any Linux task, but there are a few complications.

The kernel, in this case spufs, has by design no knowledge about what program it is running, the user space program can simply load anything into local storage. In order for oprofile to work, a new “object-id” file was added to spufs, which is used by libspe2 to tell oprofile the location of the executable in the process address space. This file is typically written when an application is first started and does not have any relevance except when profiling.

Oprofile uses the object-id in order to map the local store addresses back to a file on the disk. This can either be a plain SPU executable file, or a PowerPC ELF file that embeds the SPU executable as a blob. This means that every sample from oprofile has three values: The offset in local store, the file it came from, and the offset in that file at which the ELF executable starts.

To make things more complicated, oprofile also needs to deal with overlays, which can have different code at the same location in local storage at different times. In order

to get these right, oprofile parses some of the ELF headers of that file in kernel space when it is first loaded, and locates an overlay table in SPE local storage with this to find out which overlay was present for each sample it took.

Another twist is self-modifying code on the SPE, which happens to be used rather frequently, e.g. in order to do system calls. Unfortunately, there is nothing that oprofile can safely do about this.

7 Combined Debugger

One of the problems with earlier version of GDB for SPU was that GDB can only operate on either the PPE or the SPE. This has now been overcome by the work of Ulrich Weigand on a combined PPE/SPE debugger.

A single GDB binary now understands both instruction sets and knows how switch between the two. When GDB looks at the state of a thread, it now checks if it is in the process of executing the spu_run system call. If not, it shows the state of the thread on the PPE side using ptrace, otherwise it looks at the SPE registers through spufs.

This can work because the SIGSTOP signal is handled similarly in both cases. When gdb sends this signal to a task running on the SPE, it returns from the spu_run system call and suspends itself in the kernel. GDB can then do anything to the context and when it sends a SIGCONT, spu_run will be restarted with updated arguments.

8 Legal Statement

Linux Kernel Debugging on Google-sized clusters

Martin Bligh
Google
mbligh@mbligh.org

Mathieu Desnoyers
École Polytechnique de Montréal
mathieu.desnoyers@polymtl.ca

Rebecca Schultz
Google
rschultz@google.com

Abstract

This paper will discuss the difficulties and methods involved in debugging the Linux kernel on huge clusters. Intermittent errors that occur once every few years are hard to debug and become a real problem when running across thousands of machines simultaneously. The more we scale clusters, the more reliability becomes critical. Many of the normal debugging luxuries like a serial console or physical access are unavailable. Instead, we need a new strategy for addressing thorny intermittent race conditions. This paper presents the case for a new set of tools that are critical to solve these problems and also very useful in a broader context. It then presents the design for one such tool created from a hybrid of a Google internal tool and the open source LTTng project. Real world case studies are included.

1 Introduction

Well established techniques exist for debugging most Linux kernel problems; instrumentation is added, the error is reproduced, and this cycle is repeated until the problem can be identified and fixed. Good access to the machine via tools such as hardware debuggers (ITPs), VGA and serial consoles simplify this process significantly, reducing the number of iterations required. These techniques work well for problems that can be reproduced quickly and produce a clear error such as an oops or kernel panic. However, there are some types of problems that cannot be properly debugged in this fashion as they are:

- Not easily reproducible on demand;
- Only reproducible in a live production environment;
- Occur infrequently, particularly if they occur infrequently on a single machine, but often enough across a thousand-machine cluster to be significant;
- Only reproducible on unique hardware; or
- Performance problems, that don't produce any error condition.

These problems present specific design challenges; they require a method for extracting debugging information from a running system that does not impact performance, and that allows a developer to drill down on the state of the system leading up to an error, without overloading them with inseparable data. Specifically, problems that only appear in a full-scale production environment require a tool that won't affect the performance of systems running a production workload. Also, bugs which occur infrequently may require instrumentation of a significant number of systems in order to catch the bug in a reasonable time-frame. Additionally, for problems that take a long time to reproduce, continuously collecting and parsing debug data to find relevant information may be impossible, so the system must have a way to prune the collected data.

This paper describes a low-overhead, but powerful, kernel tracing system designed to assist in debugging this class of problems. This system is lightweight enough to run on production systems all the time, and allows for an arbitrary event to trigger trace collection when the bug occurs. It is capable of extracting only the information leading up to the bug, provides a good starting point for analysis, and it provides a framework for easily adding more instrumentation as the bug is tracked. Typically the approach is broken down into the following stages:

1. Identify the problem – for an error condition, this is simple; however, characterization may be more difficult for a performance issue.
2. Create a trigger that will fire when the problem occurs – it could be the error condition itself, or a timer that expires.

   - Use the trigger to dump a buffer containing the trace information leading up to the error.
   - Log the trigger event to the trace for use as a starting point for analysis.
3. Dump information about the succession of events leading to the problem.
4. Analyze results.

In addition to the design and implementation of our tracing tool, we will also present several case studies illustrating the types of errors described above in which our tracing system proved an invaluable resource.

After the bug is identified and fixed, tracing is also extremely useful to demonstrate the problem to other people. This is particularly important in an open source environment, where a loosely coupled team of developers must work together without full access to each other's machines.

2 Related Work

Before being used widely in such large-scale contexts, kernel tracers have been the subject of a lot of work in the past. Besides each and every kernel programmer writing his or her own ad-hoc tracer, a number of formalized projects have presented tracing systems that cover some aspect of kernel tracing.

Going through the timeline of such systems, we start with the Linux Trace Toolkit [6] which aimed primarily at offering a kernel tracing infrastructure to trace a static, fixed set of important kernel-user events useful to understand interactions between kernel and user-space. It also provided the ability to trace custom events. User-space tracing was done through device write. Its high-speed kernel-to-user-space buffering system for extraction of the trace data led to the development of RelayFS [3], now known as Relay, and part of the Linux kernel.

The K42 [5] project, at IBM Research, included a kernel and user-space tracer. Both kernel and user-space applications write trace information in a shared memory segment using a lockless scheme. This has been ported to LTT and inspired the buffering mechanism of LTTng [7], which will be described in this paper.

The SystemTAP[4] project has mainly been focused on providing tracing capabilities to enterprise-level users for diagnosing problems on production systems. It uses the kprobes mechanism to provide dynamic connection of probe handlers at particular instrumentation sites by insertion of breakpoints in the running kernel. SystemTAP defines its own probe language that offers the security guarantee that a programmer's probes won't have side-effects on the system.

Ingo Molnar's IRQ latency tracer, Jens Axboe's blktrace, and Rick Lindsley's schedstats are examples of in-kernel single-purpose tracers which have been added to the mainline kernel. They provide useful information about the system's latency, block I/O, and scheduler decisions.

It must be noted that tracers have existed in proprietary real-time operating systems for years—for example, take the WindRiver Tornado (now replaced by LTTng in their Linux products). Irix has had an in-kernel tracer for a long time, and Sun provides Dtrace[1], an open source tracer for Solaris.

3 Why do we need a tracing tool?

Once the cause of a bug has been identified, fixing it is generally trivial. The difficulty lies in making the connection between an error conveyed to the user—an oops, panic, application error—and the source. In a complex, multi-threaded system such as the Linux kernel, which is both reentrant and preemptive, understanding the paths taken through kernel code can be difficult, especially where the problem is intermittent (such as a race condition). These issues sometimes require powerful information gathering and visualization tools to comprehend.

Existing solutions, such as statistical profiling tools like oprofile, can go some way to presenting an overall view of a system's state and are helpful for a wide class of problems. However, they don't work well for all situations. For example, identifying a race condition requires capturing the precise sequence of events that occurred; the tiny details of ordering are what is needed to identify the problem, not a broad overview. In these situations, a tracing tool is critical. For performance issues, tools like OProfile are useful for identifying hot functions, but don't provide much insight into intermittent latency problems, such as some fraction of a query taking 100 times as long to complete for no apparent reason.

Often the most valuable information for identifying these problems is in the state of the system preceding the event. Collecting that information requires continuous logging and necessitates preserving information about the system for at least some previous section of time.

In addition, we need a system that can capture failures at the earliest possible moment; if a problem takes a week to reproduce, and 10 iterations are required to collect enough information to fix it, the debugging process quickly becomes intractable. The ability to instrument a wide spectrum of the system ahead of time, and provide meaningful data the first time the problem appears, is extremely useful. Having a system that can be deployed in a production environment is also invaluable. Some problems only appear when you run your application in a full cluster deployment; re-creating them in a sandbox is impossible.

Most bugs seem obvious in retrospect, after the cause is understood; however, when a problem first appears, getting a general feel for the source of the problem is essential. Looking at the case studies below, the reader may be tempted to say "you could have detected that using existing tool X;" however, that is done with the benefit of hindsight. It is important to recognize that in some cases, the bug behavior provides no information about what subsystem is causing the problem or even what tools would help you narrow it down. Having a single, holistic tracing tool enables us to debug a wide variety of problems quickly. Even if not all necessary sites are instrumented prior to the fact, it quickly identifies the general area the problem lies in, allowing a developer to quickly and simply add instrumentation on top of the existing infrastructure.

If there is no clear failure event in the trace (e.g. an OOM kill condition, or watchdog trigger), but a more general performance issue instead, it is important to be able to visualize the data in some fashion to see how performance changes around the time the problem is observed. By observing the elapsed time for a series of calls (such as a system call), it is often easy to build an expected average time for an event making it possible to identify outliers. Once a problem is narrowed down to a particular region of the trace data, that part of the trace can be more closely dissected and broken down into its constituent parts, revealing which part of the call is slowing it down.

Since the problem does not necessarily present itself at each execution of the system call, logging data (local variables, static variables) when the system call executes can provide more information about the particularities of an unsuccessful or slow system call compared to the normal behavior. Even this may not be sufficient—if the problem arises from the interaction of other CPUs or interrupt handlers with the system call, one has to look at the trace of the complete system. Only then can we have an idea of where to add further instrumentation to identify the code responsible for a race condition.

4 Case Studies

4.1 Occasional poor latency for I/O write requests

Problem Summary: The master node of a large-scale distributed system was reporting occasional timeout errors on writes to disk, causing a cluster fail-over event. No visible errors or detectable hardware problems seemed to be related.

Debugging Approach: By setting our tracing tool to log trace data continuously to a circular buffer in memory, and stopping tracing when the error condition was detected, we were able to capture the events preceding the problem (from a point in time determined by the buffer size, e.g. 1GB of RAM) up until it was reported as a timeout. Looking at the start and end times for write requests matching the process ID reporting the timeout, it was easy to see which request was causing the problem.

By then looking at the submissions and removals from the IO scheduler (all of which are instrumented), it was obvious that there was a huge spike in IO traffic at the same time as the slow write request. Through examining the process ID which was the source of the majority of the IO, we could easily see the cause, or as it turned out in this case, two separate causes:

1. An old legacy process left over from 2.2 kernel era that was doing a full `sync()` call every 30s.
2. The logging process would occasionally decide to rotate its log files, and then call `fsync()` to make sure it was done, flushing several GB of data.

Once the problem was characterized and understood, it was easy to fix.

1. The sync process was removed, as its duties have been taken over in modern kernels by pdflush, etc.

2. The logging process was set to rotate logs more often and in smaller data chunks; we also ensured it ran in a separate thread, so as not to block other parts of the server.

Application developers assumed that since the individual writes to the log files were small, the fsync would be inexpensive; however, in some cases the resulting `fsync` was quite large.

This is a good example of a problem that first appeared to be kernel bug, but was in reality the result of a user-space design issue. The problem occurred infrequently, as it was only triggered by the fsync and sync calls coinciding. Additionally, the visibility that the trace tool provided into system behavior enabled us to make general latency improvements to the system, as well as fixing the specific timeout issue.

4.2 Race condition in OOM killer

Problem summary: In a set of production clusters, the OOM killer was firing with an unexpectedly high frequency and killing production jobs. Existing monitoring tools indicated that these systems had available memory when the OOM condition was reported. Again this problem didn't correlate with any particular application state, and in this case there was no reliable way to reproduce it using a benchmark or load test in a controlled environment.

While the rate of OOM killer events was statistically significant across the cluster, it was too low to enable tracing on a single machine and hope to catch an event in a reasonable time frame, especially since some amount of iteration would likely be required to fully diagnose the problem. As before, we needed a trace system which could tell us what the state of the system was in the time leading up to a particular event. In this case, however, our trace system also needed to be lightweight and safe enough to deploy on a significant portion of a cluster that was actively running production workloads. The effect of tracing overhead needed to be imperceptible as far as the end user was concerned.

Debugging Approach: The first step in diagnosing this problem was creating a trigger to stop tracing when the OOM killer event occurred. Once this was in place we waited until we had several trace logs to examine. It was apparent that we were failing to scan or successfully reclaim a suitable number of pages, so we instrumented the main reclaim loop. For each pass over the LRU list, we recorded the reclaim priority, the number of pages scanned, the number of pages reclaimed, and kept counters for each of 33 different reasons why a page might fail to be reclaimed.

From examining this data for the PID that triggered the OOM killer, we could see that the memory pressure indicator was increasing consistently, forcing us to scan increasing number of pages to successfully reclaim memory. However, suddenly the indicator would be set back to zero for no apparent reason. By backtracking and examining the events for all processes in the trace, we were able to determine see that a different process had reclaimed a different class of memory, and then set the global memory pressure counter back to zero.

Once again, with the problem fully understood, the bug was easy to fix through the use of a local memory pressure counter. However, to send the patch back upstream into the mainline kernel, we first had to convince the external maintainers of the code that the problem was real. Though they could not see the proprietary application, or access the machines, by showing them a trace of the condition occurring, it was simple to demonstrate what the problem was.

4.3 Timeout problems following transition from local to distributed storage

Problem summary: While adapting Nutch/Lucene to a clustered environment, IBM transitioned the filesystem from local disk to a distributed filesystem, resulting in application timeouts.

The software stack consisted of the Linux kernel, the open source Java application Nutch/Lucene, and a distributed filesystem. With so many pieces of software, the number and complexity of interactions between components was very high, and it was unclear which layer was causing the slowdown. Possibilities ranged from sharing filesystem data that should have been local, to lock contention within the filesystem, with the added possibility of insufficient bandwidth.

Identifying the problem was further complicated by the nature of error handling in the Nutch/Lucene application. It consists of multiple monitor threads running periodically to check that each node is executing properly. This separated the error condition, a timeout, from the root cause. It can be especially challenging to find the source of such problems as they are seen only in relatively long tests, in this case of 15 minutes or more. By the time the error condition was detected, its cause is no longer apparent or even observable: it has passed out of scope. Only by examining the complete execution window of the timeout—a two-minute period, with many threads—can one pinpoint the problem.

Debugging Approach: The cause of this slowdown was identified using the LTTng/LTTV tracing toolkit. First, we repeated the test with tracing enabled on each node, including the user-space application. This showed that the node triggering the error condition varied between runs. Next, we examined the trace from this node at the time the error condition occurred in order to learn what happened in the minutes leading up to the error. Inspecting the source code of the reporting process was not particularly enlightening, as it was simply a monitoring process for the whole node. Instead, we had to look at the general activity on this node; which was the most active thread, and what was it doing?

The results of this analysis showed that the most active process was doing a large number of `read` system calls. Measuring the duration of these system calls, we saw that each was taking around 30ms, appropriate for disk or network access, but far too long for reads from the data cache. It thus became apparent that the application was not properly utilizing its cache; increasing the cache size of the distributed system completely resolved the problem.

This problem was especially well suited to an investigation through tracing. The timeout error condition presented by the program was a result of a general slowdown of the system, and as such would not present with any obvious connection with the source of the problem. The only usable source of information was the two-minute window in which the slowdown occurred. A trace of the interactions between each thread and the kernel during this window revealed the specific execution mode responsible for the slowdown.

4.4 Latency problem in printk on slow serialization

Problem Summary: User-space applications randomly suffer from scheduler delays of about 12ms.

While some problems can be blamed on user-space design issues that interact negatively with the kernel, most user-space developers expect certain behaviors from the kernel and unexpected kernel behaviors can directly and negatively impact user-space applications, even if they aren't actually errors. For instance, [2] describes a problem in which an application sampling video streams at 60Hz was dropping frames. At this rate, the application must process one frame every 16.6ms to remain synchronized with incoming data. When tracing the kernel timer interrupt, it became clear that delays in the scheduler were causing the application to miss samples. Particularly interesting was the jitter in timer interrupt latency as seen in Figure 1.

A normal timer IRQ should show a jitter lower than the actual timer period in order to behave properly. However, tracing showed that under certain conditions, the timing jitter was much higher than the timer interval. This was first observed around tracing start and stop. Some timer ticks, accounting for 12ms, were missing (3 timer ticks on a 250HZ system).

Debugging Approach: Instrumenting each `local_irq_{save,restore,disable,enable}` macro provided the information needed to find the problem, and extracting the instruction pointer at each call to these macros revealed exactly which address disabled the interrupts for too long around the problematic behavior.

Inspecting the trace involved first finding occurrences of the problematic out-of-range intervals of the interrupt timer and using this timestamp to search backward for the last `irq_save` or `irq_disable` event. Surprisingly, this was `release_console_sem` from `printk`. Disabling the serial console output made the problem disappear, as evidenced by Figure 2. Disabling interrupts while waiting for the serial port to flush the buffers was responsible for this latency, which not only affects the scheduler, but also general timekeeping in the Linux kernel.

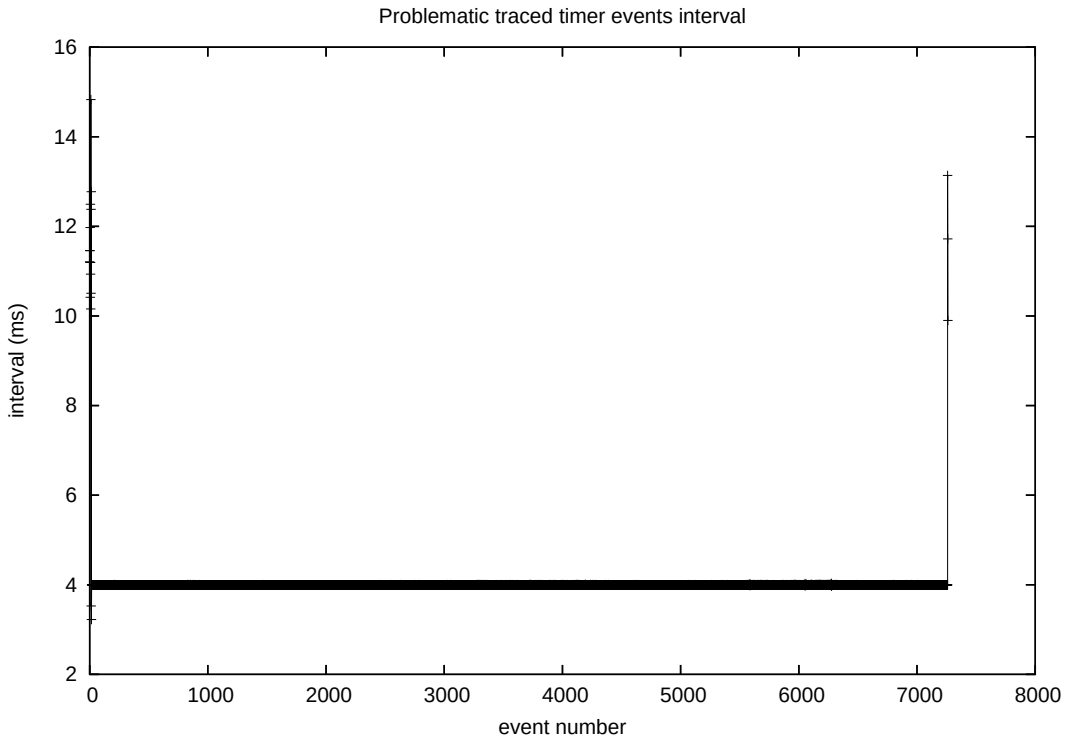


Figure 1: Problematic traced timer events interval

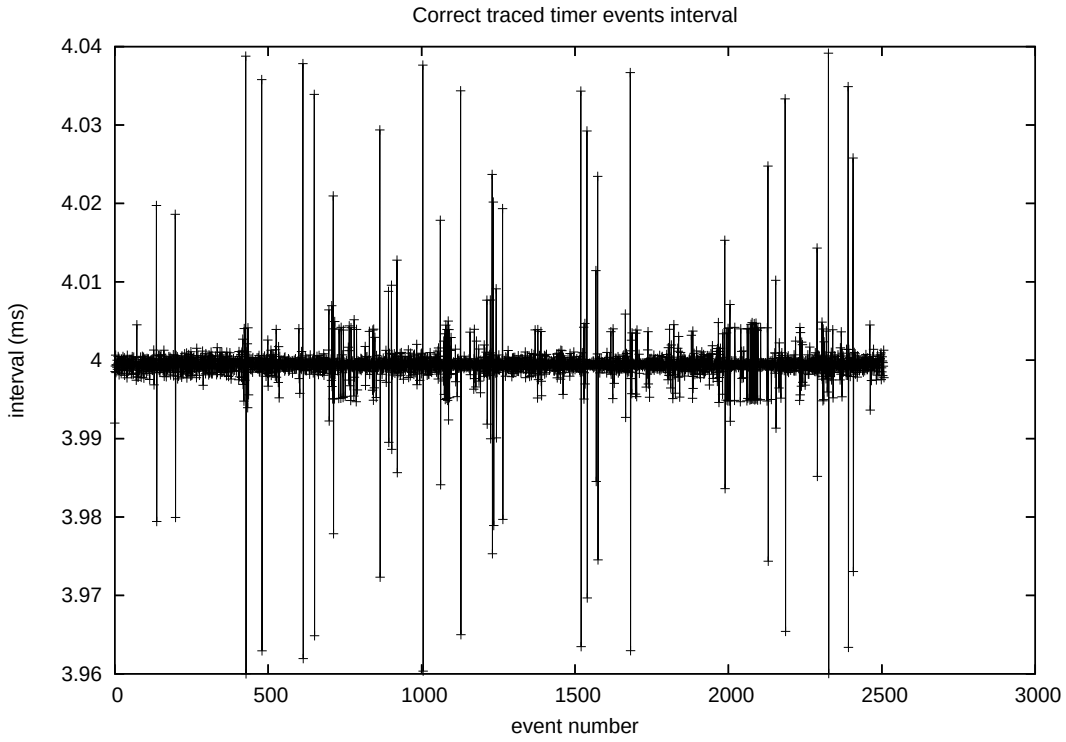


Figure 2: Correct traced timer events interval

4.5 Hardware problems causing a system delay

Problem Summary: The video/audio acquisition software running under Linux at Autodesk, while in development, was affected by delays induced by the PCI-Express version of a particular card. However, the manufacturer denied that their firmware was the cause of the problem, and insisted that the problem was certainly driver or kernel-related.

Debugging Approach: Using LTTng/LTTV to trace and analyze the kernel behavior around the experienced delay led to the discovery that this specific card's interrupt handler was running for too long. Further instrumentation within the handler permitted us to pinpoint the problem more exactly—a register read was taking significantly longer than expected, causing the deadlines to be missed for video and audio sampling. Only when confronted with this precise information did the hardware vendor acknowledge the issue, which was then fixed within a few days.

5 Design and Implementation

We created a hybrid combination of two tracing tools—Google's Ktrace tool and the open source LTTng tool, taking the most essential features from each, while trying to keep the tool as simple as possible. The following set of requirements for tracing was collected from users and from experience through implementation and use:

- When not running, must have zero effective impact.
- When running, should have low enough impact so as not to disturb the problem, or impede production traffic.
- Spooling data off the system should not completely saturate the network.
- Compact data format—must be able to store large amounts of data using as little storage as possible.
- Applicability to a wide range of kernel points, i.e., able to profile in interrupt context, and preferably in NMI context.
- User tools should be able to read multiple different kernel versions, deal with custom debug points, etc.
- One cohesive mechanism (and time ordered stream), not separate tools for scheduler, block tracing, VM tracing, etc.

The resulting design has four main parts described in detail in the sections that follow:

1. a logging system to collect and store trace data and make it available in user-space;
2. a triggering system to identify when an error has occurred and potentially stop tracing;
3. an instrumentation system that meets the performance requirements and also is easily extensible; and
4. an analysis tool for viewing and analyzing the resulting logs.

5.1 Collection and Logging

The system must provide buffers to collect trace data whenever a trace point is encountered in the kernel and have a low-overhead mechanism for making that data available in user-space. To do this we use preallocated, per-CPU buffers as underlying data storage and fast data copy to user-space performed via Relay. When a "trigger" event occurs, assuming the machine is still in a functional state, passing data to user-space is done via simple tools reading the Relay interfaces. If the system has panicked, we may need to spool the data out over the network to another machine (or to local disk), as in the the netdump or crashdump mechanisms.

The in-kernel buffers can be configured to operate in three modes:

- Non-overwrite – when the buffer is full, drop events and increment an event lost counter.
- Overwrite – use the buffer as a circular log buffer, overwriting the oldest data.
- Hybrid – a combination of the two where high rate data is overwritten, but low rate state information is treated as non-overwrite.

Each trace buffer actually consists of a group of per-cpu buffers, each assigned to high, medium, and low rate data. High-rate data accounts for the most common event types described in detail below—system call entry and exits, interrupts, etc. Low-rate data is generally static throughout the trace run and consists in part of the information required to decode the resulting trace, system data type sizes, alignment, etc. Medium-rate channels record meta-information about the system, such as the mapping of interrupt handlers to devices (which might change due to Hotplug), process names, their memory maps, and opened file descriptors. Loaded modules and network interfaces are also treated as medium-rate events. By iterating on kernel data structures we can record a listing of the resources present at trace start time, and update it whenever it changes, thus building a complete picture of the system state.

Separating high-rate events (prone to fill the buffers quickly) from lower rate events allows us to use the maximum space for high-rate data without losing the valuable information provided by the low- and medium-rate channel. Also, it makes it easy to create a hybrid mode system where the last few minutes of interrupt or system call information can be viewed, and we can also get the mapping of process IDs to names even if they were not created within that time window.

Multiple channels can also be used to perform fast user-space tracing, where each process is responsible for writing the trace to disk by itself without going through a system call and Xen hypervisor tracing. The trace merging is performed by the analysis tool in the same manner in which the multiple CPU buffers are handled, permitting merging the information sources at post-processing time.

It may also be useful to integrate other forms of information into the trace, in order to get one merged stream of data—i.e., we could record readprofile-style data (where the instruction pointer was at a given point in time) either in the timer tick event, or as a periodic dump of the collated hash table data. Also functions to record meminfo, slabinfo, ps data, user-space and kernel stacks for the running threads might be useful, though these would have to be enabled on a custom basis. Having all the data in one place makes it significantly easier to write analysis and visualization tools.

5.2 Triggering

Often we want to capture the state of the system in a short period of time preceding a critical error or event. In order to avoid generating massive amounts of data and the performance impact of disk or network writes to the system, we leave the system logging into a circular buffer, then stop tracing when the critical event occurs.

To do this, we need to create a trigger. If this event can easily be recognized by a user-space daemon, we can simply call the usual tracing interface with an instruction to stop tracing. For some situations, a small in-kernel trigger is more appropriate. Typical trigger events we have used include:

- OOM kill;
- Oops / panic;
- User-space locks up (processes are not getting scheduled);
- User application indicates poor response from system; or
- Manual intervention from user.

5.3 Instrumentation

When an instrumentation point is encountered, the tracer takes a timestamp and the associated event data and logs it to our buffers. Each encountered instrumentation point must have minimum overhead, while providing the most information.

Section 5.3.1 explains how our system minimizes the impact of instrumentation and compares and contrasts static and dynamic instrumentation schemes.

We will discuss the details of our event formats in Section 5.3.2 and our approach to timestamping in Section 5.3.3.

To eliminate cache-line bouncing and potential race conditions, each CPU logs data to its own buffer, and system-wide event ordering is done via timestamps. Because we would like to be able to instrument reentrant contexts, we must provide a locking mechanism to avoid potential race conditions. We have investigated two options described in Section 5.3.4.

5.3.1 Static vs. Dynamic Instrumentation Points

There are two ways we can insert trace points—at static markers that are pre-defined in the source code, or dynamically insert them while the system is running. For standard events that we can anticipate the need for in advance, the static mechanism has several advantages. For events that are not anticipated in advance, we can either insert new static points in the source code, compile a new kernel and reboot, or insert dynamic probes via a mechanism such as kprobes. Static vs dynamic markers are compared below:

- Trace points from static markers are significantly faster in use. Kprobes uses a slow int3 mechanism; development efforts have been made to create faster dynamic mechanisms, but they are not finished, very complex, cannot instrument fully preemptible kernels, and they are still significantly slower than static tracing.
- Static trace points can be inserted anywhere in the code base; dynamic probes are limited in scope.
- Dynamic trace points cannot easily access local variables or registers at arbitrary points within a function.
- Static trace points are maintained within the kernel source tree and can follow its evolution; dynamic probes require constant maintenance outside of the tree, and new releases if the traced code changes. This is more of a problem for kernel developers, who mostly work with mainline kernels that are constantly changing.
- Static markers have a potential performance impact when not being used—with care, they can be designed so that this is practically non-existent, and this can be confirmed with performance benchmarks.

We use a marker infrastructure which is a hook-callback mechanism. Hooks are our markers placed in the kernel at the instrumentation site. When tracing is enabled, these are connected to the callback probes—the code executed to perform the tracing. The system is designed to have an impact as low as possible on the system performance, so markers can be compiled into a production kernel without appreciable performance impact. The probe callback connection to its markers is done dynamically. A predicted branch is used to skip the hook stack setup and function call when the marker is "disabled" (no probe is connected). Further optimizations can be implemented for each architecture to make this branch faster.

The other key facet of our instrumentation system is the ability to allow the user to extend it. It would be impossible to determine in advance the complete set of information that would be useful for a particular problem, and recording every thing occurring on a system would be clearly be impractical if not infeasible. Instead, we have designed a system for adding instrumentation iteratively from a coarse-grained level including major events like system calls, scheduling, interrupts, faults, etc. to a finer grained level including kernel synchronization primitives and important user-space functions. Our tool is capable of dealing with an extensible set of user-definable events, including merged information coming from both kernel and user-space execution contexts, synchronized in time.

Events can also be filtered; the user can request which event types should be logged, and which should not. By filtering only by event type, we get an effective, if not particularly fine-grained filter, and avoid the concerns over inserting buggy new code into the kernel, or the whole new languages that tools like Dtrace and Systemtap invent in order to fix this problem. In essence, we have chosen to do coarse filtering in the kernel, and push the rest of the task to user-space. This design is backed up by our efficient probes and logging, compact logging format, and efficient data relay mechanism to user-space (Relay).

5.3.2 Event Formats

It would be beneficial to log as much data about the system state as possible, but instrumenting every interrupt or system call clearly will rapidly generate large volumes of data. To maximize the usefulness of our tool, we must store our event data in the most efficient way possible. In Google's ktrace tool, for the sake of compactness and alignment we chose to make our most common set of events take up 8 bytes. The best compromise between data compactness and information completeness within these bytes was to use the first 4 bytes for type and timestamp information, and the second 4 for an

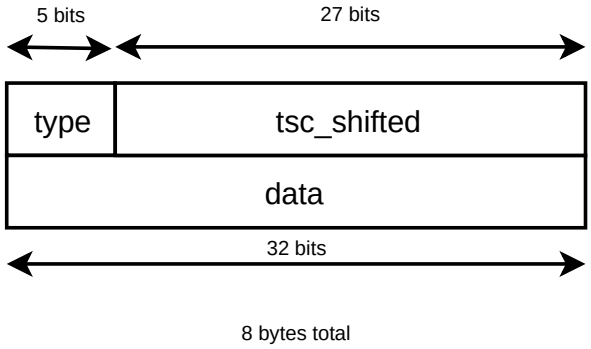


Figure 3: Common event format

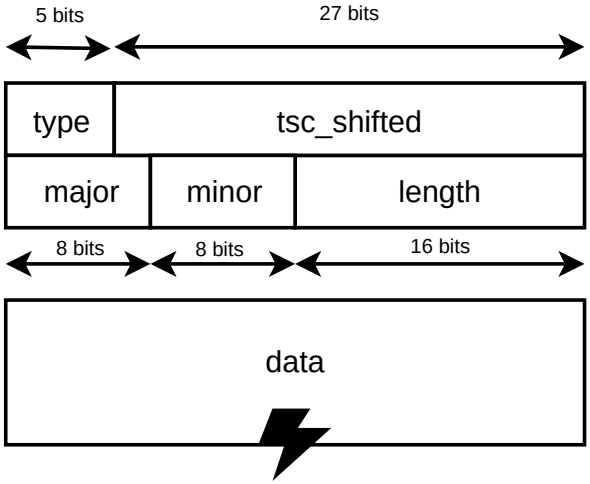


Figure 4: Expanded event format

event-specific data payload. The format of our events is shown in Figure 3.

Commonly logged events include:

- System call entry / exit (including system call number, lower bytes of first argument)
- Interrupt entry / exit
- Schedule a new task
- Fork / exec of a task, new task seen
- Network traffic
- Disk traffic
- VM reclaim events

In addition to the basic compact format, we required a mechanism for expanding the event space and logging data payloads larger than 4 bytes. We created an expanded event format, shown in Figure 4, that can be used to store larger events needing more data payload space (up to 64K). The normal 32-bit data field is broken into a major and minor expanded event types (256 of each) and a 16-bit length field specifying the length of the data payload that follows.

LTTng's approach is similar to Ktrace; we use 4-byte event headers, followed by a variable size payload. The compact format is also available; it records the timestamp, the event ID, and the payload in 4 bytes. It dynamically calculates the minimum number of bits required to represent the TSC and still detect overflows. It uses the timer frequency and CPU frequency to determine this value.

5.3.3 Timestamps

Our instrumentation system must provide an accurate, low-overhead timestamp to associate with each logged event. The ideal timestamp would be a high-resolution fixed frequency counter, that has very low cost to retrieve, is always monotonic, and is synchronized across all CPUs and readable from both kernel and user-space. However, due to the constraints of current hardware, we are forced to an uncomfortable compromise.

If we look at a common x86-style architecture (32- or 64-bit), choices of time source include PIT, TSC, and HPET. The only time source with acceptable overhead is TSC; however, it is not constant frequency, or well synchronized across platforms. It is also too high-frequency to be compactly logged. The chosen compromise has been to log the TSC at every event, truncated (both on the left and right sides)—effectively, in Ktrace:

$$tsc_{timestamp} = (tsc >> 10)\&(2^{27})$$

On a 2GHz processor, this gives an effective resolution of 0.5us, and takes 27 bits of space to log. LTTng calculates the shifting required dynamically.

However, this counter will roll over every 128 seconds. To ensure we can both unroll this information properly and match it up to the wall time (e.g. to match user-space events) later, we periodically log a timestamp event:

A new timestamp event must be logged:

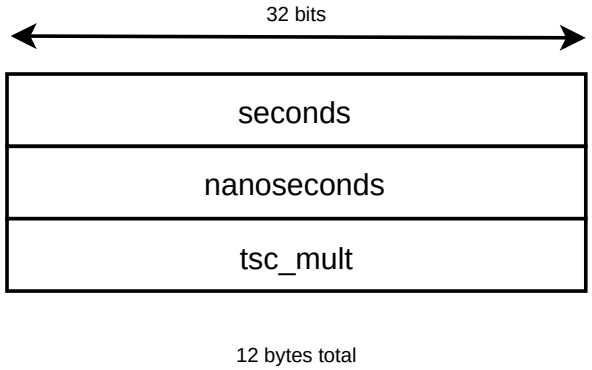


Figure 5: Timestamp format

1. More frequently than the logged timestamp derived from the TSC rolls over.
2. Whenever TSC frequency changes.
3. Whenever TSCs are resynchronized between CPUs.

The effective time of an event is derived by comparing the event TSC to the TSC recorded in the last timestamp and multiplying by a constant representing the current processor frequency.

$$\delta_{walltime} = (event_{tsc} - timestamp_{tsc}) * k_{tsc\_freq}$$

$$event_{walltime} = \delta_{walltime} + timestamp_{walltime}$$

5.3.4 Locking

One key design choice for the instrumentation system for this tool was how to handle potential race conditions from reentrant contexts. The original Google tool, Ktrace, protected against re-entrant execution contexts by disabling interrupts at the instrumentation site, while LTTng uses a lock-less algorithm based on atomic operations local to one CPU (`asm/local.h`) to take timestamps and reserve space in the buffer. The atomic method is more complex, but has significant advantages—it is faster, and it permits tracing of code paths reentering even when IRQs are disabled (lockdep lock dependency checker instrumentation and NMI instrumentation are two examples where is has shown to be useful). The performance improvement of using atomic operations (local compare-and-exchange: 9.0ns) instead of disabling interrupts (save/restore: 210.6ns) on a 3GHz Pentium 4 removes 201.6ns from each probe's execution time. Since the average probe duration of LTTng is about 270ns in total, this is a significant performance improvement.

The main drawback of the lock-less scheme is the added code complexity in the buffer-space reservation function. LTTng's reserve function is based on work previously done on the K42 research kernel at IBM Research, where the timestamp counter read is done within a compare-and-exchange loop to insure that the timestamps will increment monotonically in the buffers. LTTng made some improvements in how it deals with buffer boundaries; instead of doing a separate timestamp read, which can cause timestamps of buffer boundaries to go backward compared to the last/first events, it computes the offsets of the buffer switch within the compare-and-exchange loop and effectively does it when the compare-and-exchange succeeds. The rest of the callbacks called at buffer switch are then called out-of-order. Our merged design considered the benefit of such a scheme to outweigh the complexity.

5.4 Analysis

There are two main usage modes for the tracing tools:

- Given an event (e.g. user-space lockup, OOM kill, user-space noticed event, etc.), we want to examine data leading up to it.
- Record data during an entire test run, sift through it off-line.

Whenever an error condition is not fatal or recurring, taking only one sample of this condition may not give a full insight into what is really happening on the system. One has to verify whether the error is a single case or periodic, and see if the system always triggers this error or if it sometimes shows a correct behavior. In these situations, recording the full trace of the systems is useful because it gives a better overview of what is going on globally on the system.

However, this approach may involve dealing with huge amounts of data, in the order of tens of gigabytes per node. The Linux Trace Toolkit Viewer (LTTV) is designed to do precisely this. It gives both a global graphical overview of the trace, so patterns can be easily identified, and permits the user to zoom into the trace to get the highest level of detail.

Multiple different user-space visualization tools have been written (in different languages) to display or process the tracing data, and it's helpful for them to share this pre-processing phase. These tools fall into two categories:

1. Text printer – one event per line, formatted in a way to make it easy to parse with simple scripts, and fairly readable by a kernel developer with some experience and context.
2. Graphical – easy visualization of large amounts of data. More usable by non-kernel-developers.

6 Future Work

The primary focus of this work has been on creating a single-node trace tool that can be used in a clustered environment, but it is still based on generating a view of the state of a single node in response to a particular trigger on that node. This system lacks the ability to track dependent events between nodes in a cluster or to follow dependencies between nodes. The current configuration functions well when the problem can be tracked to a single node, but doesn't allow the user to investigate a case where events on another system caused or contributed to an error. To build a cluster-wide view, additional design features would be needed in the triggering, collection, and analysis aspects of the trace tool.

- Ability to start and stop tracing on across an entire cluster when a trigger event occurs on one node.
- Low-overhead method for aggregating data over the network for analysis.
- Sufficient information to analyze communication between nodes.
- A unified time base from which to do such analysis.
- An analysis tool capable of illustrating the relationships between systems and displaying multiple parallel traces.

Relying on NTP to provide said synchronization appears to be too imprecise. Some work has been started in this area, primarily aiming at using TCP exchanges between nodes to synchronize the traces. However, it is restrained to a limited subset of network communication: it does not deal with UDP and ICMP packets.

References

[1] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX '04*, 2004.

[2] Mathieu Desnoyers and Michel Dagenais. Low disturbance embedded system tracing with linux trace toolkit next generation. In *ELC (Embedded Linux Conference) 2006*, 2006.

[3] Mathieu Desnoyers and Michel Dagenais. The lttng tracer : A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium) 2006*, pages 209–224, 2006.

[4] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In *OLS (Ottawa Linux Symposium) 2005*, 2005.

[5] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing, 2003 ACM/IEEE Conference*, 2003.

[6] Karim Yaghmour and Michel R. Dagenais. The linux trace toolkit. *Linux Journal*, May 2000.

[7] Tom Zanussi, Karim Yaghmour Robert Wisniewski, Richard Moore, and Michel Dagenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *OLS (Ottawa Linux Symposium) 2003*, pages 519–531, 2003.

Ltrace Internals

Rodrigo Rubira Branco
IBM
rrbranco@br.ibm.com

Abstract

ltrace is a program that permits you to track runtime library calls in dynamically linked programs without recompiling them, and is a really important tool in the debugging arsenal. This article will focus in how it has been implemented and how it works, trying to cover the actual lacks in academic and in-deep documentation of how this kind of tool works (setting the breakpoints, analysing the executable/library symbols, interpreting elf, others).

1 Introduction

ltrace is divided into many source files; some of these contain architecture-dependent code, while some others are generic implementations.

The idea is to go through the functions, explaining what each is doing and how it works, beginning from the entry point function, `main`.

2 int main(int argc, char **argv) – ltrace.c

The `main` function sets up ltrace to perform the rest of its activities.

It first sets up the terminal using the `guess_cols()` function that tries to ascertain the number of columns in the terminal so as to display the information output by ltrace in an ordely manner. The column count is initially queried from the `$COLUMNS` environment variable (if that is not set, the TIOCGWINSZ ioctl is used instead). Then the program options are handled using the `process_options()` function to processes the ltrace command line arguments, using the `getopt()` and `getopt_long()` functions to parse them.

It then calls the `read_config_file()` function on two possible configuration files.

It calls `read_config_file()` first with SYSCONFDIR's `ltrace.conf` file. If `$HOME` is set, it then calls the function with `$HOME/.ltrace.conf`. This function opens the specified file and reads in from it line-by-line, sending each line to the `process_line()` function to verify the syntax of the config file based on the line supplied to it. It then returns a function structure based on the function information obtained from said line.

If `opt_e` is set, then a list is output by the `debug()` function.

If passed a command invocation, `ltrace` will execute it via the `execute_program()` function which takes the return value of the `open_program()` function as an argument.

Ltrace will attached to any supplied pids using the `open_pid()` function.

At the end of this function the `process_event()` function is called in an infinite loop, receiving the return value of the `wait_for_something()` function as its argument.

3 struct process *open_program(char *filename, pid_t pid) – proc.c

This function implements a number of important tasks needed by ltrace. `open_program` allocates a process structure's memory and sets the filename and pid (if needed), adds the process to the linked-list of processes traced by ltrace, and most importantly initalizes breakpoints by calling `breakpoints_init()`.

4 void breakpoints_init(struct process *proc) – breakpoints.c

The `breakpoints_init()` function is responsible for setting breakpoints on every symbol in the program being traced. It calls the `read_elf()` function

which returns an array of `library_symbol` structures, which it processes based on `opt_e`. Then it iterates through the array of `library_symbol` structures and calls the `insert_breakpoint()` function on each symbol.

5 struct library_symbol *read_elf(struct process *proc) – elf.c

This function retrieves a process's list of symbols to be traced. It calls `do_init_elf()` on the executable name of the traced process and for each library supplied by the `-l` option. It loops across the PLT information found therein.

For each symbol in the PLT information, a `GElf_Rel` structure is returned by a call to `gelf_getrel()`, if the `d_type` is `ELF_T_REL` and `gelf_getrela()` if not. If the return value of this call is NULL, or if the value returned by `ELF64_R_SYM(rela.r_info)` is greater than the number of dynamic symbols or the `rela.r_info` symbol is not found, then the function calls the `error()` function to exit the program with an error.

If the symbol value is NULL and the `PLTs_initialized_by_here` flag is set, then the `need_to_reinitialize_breakpoints` member of the `proc` structure is set.

The name of the symbol is calculated and this is passed to a call to `in_load_libraries()`. If this returns a positive value, then the symbol address is calculated via the `arch_plt_sym_val()` function and the `add_library_symbol()` function is called to add the symbol to the `library_symbols` list of dynamic symbols. At this point if the `need_to_reinitialize_breakpoints` member of the proc structure is set, then a `pt_e_t` structure `main_cheat` is allocated and its values are set. After this a loop is made over the `opt_x` value (passed by the `-x` option) and if the `PLTs_initialized_by_here` variable matches the name of one of the values, then `main_cheat` is freed and the loop is broken. If no match is found, then `opt_x` is set to the final value of `main_cheat`.

A loop is then made over the symtab, or symbol table variable. For each symbol `gelf_getsym()` is called, which if it fails provokes ltrace to exit with an error message via the `error()` function. A nested loop is then made over the values passed to `opt_x` via the `-x` option. For each value a comparison is made against the name of each symbol. If there is a match, then the symbol is added to the `library_symbols` list via `add_library_symbol()` and the nested loop breaks.

At the end of this loop a final loop is made over the values passed to `opt_x` via the `-x` option.

For each value with a valid name member a comparison is made to the `E_ENTRY_NAME` value, which represents the program's entry point. If this comparison should prove true, then the symbol is entered into the `library_symbols` list via `add_library_symbol()`.

At the end of the function, any libraries passed to ltrace via the `-l` option are closed via the `do_close_elf()` function[1] and the `library_symbols` list is returned.

6 static void do_init_elf(struct ltelf *lte, const char *filename) – elf.c

The passed `ltelf` structure is set to zero and `open()` is called to open the passed filename as a file. If this fails, then ltrace exits with an error message. The `elf_begin()` function is then called, following which various checks are made via `elf_kind()` and `gelf_getehdr()`. The type of the elf header is checked so as to only process executable files or dynamic library files.

If the file is not of one of these types, then ltrace exits with an error. Ltrace also exits with an error if the elf binary is from an unsupported architecture.

The ELF section headers are iterated over and the `elf_getscn()` function is called, then the variable *name* is set via the `elf_strptr()` function (if any of the above functions fail, ltrace exits with an error message).

A comparison is then made against the section header type and the data for it is obtained via a call to `elf_getdata()`.

[1] This function is called to close open ELF images. A check is made to see if the `ltelf` structure has an associated hash value allocated and if so this hash value is deallocated via a call to `free()`. After this `elf_end()` is called and the file descriptor associated with the image is closed.

For SHT_DYNSYM (dynamic symbols), the lte->dynsym is filled via a call to elf_getdata(), where the dynsym_count is calcuated by dividing the section header size by the size of each entry. If the attempt to get the dynamic symbol data fails, ltrace exits with an error message. The elf_getscn() function is then called, passing the section header sh_link variable. If this fails, then ltrace exits with an error message. Using the value returned by elf_getscn(), the gelf_getshdr() function is called and if this fails, ltrace exits with an error message.

For SHT_DYNAMIC an Elf_Data structure *data* is set via a call to elf_getdata() and if this fails, ltrace exits with an error message. Every entry in the section header is iterated over and the following occurs: The gelf_getdyn() function is called to retrieve the .dynamic data and if this fails, ltrace exits with an error message; relplt_addr and relplt_size are calculated from the returned dynamic data.

For SHT_HASH values an Elf_Data structure *data* is set via a call to elf_getdata() and if this fails, ltrace exits with an error message. If the entry size is 4 then lte->hash is simply set to the dynamic data buf data->d_buf. Otherwise it is 8. The correct amount of memory is allocated via a call to malloc and the hash data into copied into lte->hash.

For SHT_PROGBITS, checks are made to see if the name value is .plt or .pd, and if so, the correct elements are set in the lte->plt_addr/lte->opd and lte->plt_size and lte->pod_size structures. In the case of OPD, the lpe->opd structure is set via a call to elf_rawdata(). If neither the dynamic symbols or the dynamic strings have been found, then ltrace exits with an error message. If relplt_addr and lte->plt_addr are non-null, the section headers are iterated across and the following occurs:

- The elf_getscn() function is called.
- If the sh_addr is equal to the relpt_addr and the sh_size matches the relplt_size (i.e., this section is the .relplt section) then lte->relplt is obtained via a call to elf_getdata() and lte->relplt_count is calculated as the size of section divided by the size of each entry. If the call to elf_getdata() fails then ltrace exits with an error message.
- If the function was unable to find the .relplt section then ltrace exits with an error message.

7 static void add_library_symbol(GElf_Addr addr, const char *name, struct library_symbol **library_symbolspp, int use_elf_plt2addr, int is_weak) – elf.c

This function allocates a library_symbol structure and inserts it into the linked list of symbols represented by the library_symbolspp variable.

The structure is allocated with a call to malloc(). The elements of this structure are then set based on the arguments passed to the function. And the structure is linked into the linked list using its *next* element.

8 static GElf_Addr elf_plt2addr(struct ltelf *lte, void *addr) – elf.c

In this function the opd member of the lte structure is checked and if it is NULL, the function returns the passed address argument as the return value. If opd is non-NULL, then following occurs:

1. An offset value is calculated by subtracting the opd_addr element of the ltr structure from the passed address.
2. If this offset is greater than the opd_size element of the lte structure then ltrace exits with an error.
3. The return value is calculated as the base address (passed as lte->opd->d_buf) plus the calculated offset value.
4. This calculated final return value is returned as a GElf_Addr variable.

9 static int in_load_libraries(const char *name, struct ltelf *lte) – elf.c

This functions checks if there are any libraries passed to ltrace as arguments to the -l option. If not, then the function immediately returns 1 (one) because there is no filtering (specified libraries) in place; otherwise, a hash is calculated for the library name arguments by way of the elf_hash() function.

For each library argument, the following occurs:

1. If the hash for this iteration is NULL the loop continues to the next iteration.

2. The *nbuckets* value is obtained and the buckets and chain values are calculated based on this value from the hash.

3. For each bucket the following occurs:

   The `gelf_getsym()` function is called to get the symbol; if this fails, then ltrace exits with an error.

   A comparison is made between the passed name and the name of the current dynamic symbol. Should there be a match, the function will return a positive value (one).

4. If the code reaches here, 0 (zero) is returned.

10 void insert_breakpoint(struct process *proc, void *addr, struct library_symbol *libsym) – breakpoints.c

The `insert_breakpoint()` function inserts a breakpoint into a process at the given address (addr). If the breakpoints element of the passed proc structure has not been set it is set by calling the `dict_init()` function.

A search is then made for the address by using the `dict_find_entry()` function. If the address is not found a breakpoint structure is allocated using `calloc()`, entered into the dict hash table using `dict_enter()`, and its elements are set.

If a pid has been passed (indicating that the process is already running), this breakpoint structure along with the pid is then passed to the `enable_breakpoint()` system-dependent function.

11 void enable_breakpoint(pid_t pid, struct breakpoint *sbp) – sysdeps/linux-gnu/breakpoint.c

The `enable_breakpoint()` function is responsible for the insertion of breakpoints into a running process using the ptrace interface.

First `PTRACE_PEEKTEXT` ptrace parameter is used to save the original data from the breakpoint location and then `PTRACE_POKETEXT` is used to copy the architecture-dependent breakpoint value into the supplied memory address. The architecture-dependent breakpoint value is found in `sysdeps/linux-gnu/*/arch.h`.

12 void execute_program(struct process *sp, char **argv) – execute-program.c

The `execute_program()` function executes a program whose name is supplied as an argument to ltrace. It `fork()`s a child, changes the UID of the running child process if necessary, calls the `trace_me()` (simply calls ptrace() using the `PTRACE_TRACEME` argument, which allows the process to be traced) function and then executes the program using `execvp()`.

13 struct event *wait_for_something(void) – wait_for_something.c

The `wait_for_something()` function literally waits for an event to occur and then handles it.

The events that it treats are: Syscalls, Systets, Exiosts, exit signals, and breakpoints. `wait_for_something()` calls the `wait()` function to wait for an event.

When it awakens it calls `get_arch_dep()` on the proc member of the event structure. If breakpoints were not enabled earlier (due to the process not yet being run) they are enabled by calling `enable_all_breakpoints()`, `trace_set_options()` and then `continue_process()` (this function simply calls `continue_after_signal()`).

In this case the event is then returned as `LT_EV_NONE` which does not receive processing.

To determine the type of event that has occurred the following algorithm is used: The `syscall_p()` function is called to detect if a syscall has been called via int 0x80 (`LT_EV_SYSCALL`) or if there has been a return-from-syscall event (`LT_EV_SYSRET`). If neither of these is true, it checks to see if the process has exited or has sent an exit signal.

If neither of these is the case and the process has not stopped, an `LT_EV_UNKNOWN` event is returned.

If process is stopped and the stop signal was not `systrap`, an `LT_EV_SIGNAL` event is returned.

If none of the above cases is found to be true, it is assumed that this was a breakpoint, and an `LT_EV_BREAKPOINT` event is returned.

14 void process_event(struct event *event) – process_event.c

The `process_event()` function receives an event structure, which is generally returned by the `wait_for_something()` function.

It calls a switch-case construct based on the `event->thing` element and processes the event using one of the following functions: `process_signal()`, `process_exit()`, `process_exit_signal()`, `process_syscall()`, `process_sysret()`, or `process_breakpoint()`.

In the case of `syscall()` or `sysret()`, it calls the `sysname()` function.

15 int syscall_p(struct process *proc, int status, int *sysnum) – sysdeps/linux-gnu/*/trace.c

This function detects if a call to or return from a system call occurred. It does this first by checking the value of EAX (on x86 platforms) which it obtains with a ptrace `PTRACE_PEEKUSER` operation.

It then checks the program's call stack, as maintained by ltrace and, checking the last stack frame, it sees if the `is_syscall` element of the `proc` structure is set, which indicates a called system call. If this is set, then *2* is returned, which indicates a `sysret` event. If not, then *1* is returned, provided that there was a value in EAX.

16 static void process_signal(struct event *event) – process_event.c

This function tests the signal. If the signal is SIGSTOP it calls `disable_all_breakpoints()`, `untrace_pid()` (this function merely calls the ptrace interface using a `PTRACE_DETACH` operation), removes the process from the list of traced processes using the `remove_proc()` function, and then calls `continue_after_signal()` (this function simply calls ptrace with a `PTRACE_SYSCALL` operation) to allow the process to continue.

In the case that signal was not SIGSTOP, the function calls the `output_line()` function to display the fact of the signal and then calls `continue_after_signal()` to allow the process to continue.

17 static void process_exit(struct event *event) – process_event.c

This function is called when a traced process exits. It simply calls `output_line()` to display that fact in the terminal and then calls `remove_proc()` to remove the process from the list of traced processes.

18 static void process_exit_signal(struct event *event) – process_event.c

This function is called when when a traced program is killed. It simply calls `output_line()` to display that fact in the terminal and then calls `remove_proc()` to remove the process from the list of traced processes.

19 static void process_syscall(struct event *event) – process_event.c

This function is called when a traced program invokes a system call. If the `-S` option has been used to run ltrace, then the `output_left()` function is called to display the syscall invocation using the `sysname()` function to find the name of the system call.

It checks if the system call will result in a fork or execute operation, using the `fork_p()` and `exec_p()` functions which test the system call against those known to trigger this behavior. If it is such a signal the `disable_all_breakpoints()` function is called.

After this `callstack_push_syscall()` is called, followed by `continue_process()`.

20 static void process_sysret(struct event *event) – process_event.c

This function is called when the traced program returns from a system call. If ltrace was invoked with the `-c` or `-T` options, the `calc_time_spent()` function is called to calculate the amount of time that was spent inside the system call.

After this the function `fork_p()` is called to test if the system call was one that would have caused a process fork. If this is true, and the `-f` option was set when running ltrace, then the `gimme_arg()` function is called to get the pid of the child and the `open_pid()` function is called to begin tracing the child. In any case, `enable_all_breakpoints()` is called.

Following this, the `callstack_pop()` function is called. Then the `exec_p()` function tests if the system call was one that would have executed another program within this process and if true, the `gimme_arg()` function is called. Otherwise the `event->proc` structure is re-initialized with the values of the new program and the `breakpoints_init()` function is called to initialize breakpoints. If `gimme_arg()` does not return zero, the `enable_all_breakpoints()` function is called.

At the end of the function the `continue_process()` function is called.

21 static void process_breakpoint(struct event *event) – process_event.c

This function is called when the traced program hits a breakpoint, or when entering or returning from a library function.

It checks the value of the `event->proc->breakpoint_being_enabled` variable to determine if the breakpoint is in the middle of being enabled, in which case it calls the `continue_enabling_breakpoint()` function and this function returns. Otherwise this function continues.

It then begins a loop through the traced program's call stack, checking if the address where the breakpoint occurred matches a return address of a called function which indicates that the process is returning from a library call.

At this point a hack allows for PPC-specific behavior, and it re-enables the breakpoint. All of the library function addresses are retrieved from the call stack and translated via the `plt2addr()` function. Provided that the architecture is `EM_PPC`, the `address2bpstruct()`[2] function is called to translate the address into a breakpoint structure. The value from the address is read via the ptrace `PTRACE_PEEK` operation and this value is compared to a breakpoint value. If they do not match, a breakpoint is inserted at the address.

If the architecture is not `EM_PPC`, then the address is compared against the address of the breakpoint previously applied to the library function. If they do not match, a breakpoint is inserted at the address.

[2]This function merely calls `dict_find_entry()` to find the correct entry in `proc->breakpoints` and returns it.

Upon leaving the PPC-dependent hack, the function then loops across callstack frames using the `callstack_pop()` function until reaching the frame that the library function has returned to which is normally a single callstack frame. Again if the `-c` or `-T` options were set, `calc_time_spent()` is called.

The `callstack_pop()` function is called one final time to pop the last callstack frame and the process' return address is set in the `proc` structure as the breakpoint address. The `output_right()` function is called to log the library call and the `continue_after_breakpoint()` function is called to allow the process to continue, following which the function returns.

If no return addresses in the callstack match the breakpoint address, the process is executing in, and not returning from a library function.

The `address2bpstruct()` function is called to translate the address into a breakpoint structure.

Provided that this was a success, the following occurs:

- The stack pointer and return address to be saved in the proc stucture are obtained using the `get_stack_pointer()` and `get_return_address()` functions.

- The `output_left()` function is called to log the library function call and the `callstack_push_symfunc()` function is called. A check is then made to see if the `PLTs_initialized_by_here` variable is set, to see if the function matches the called library function's symbol name and to see if the `need_to_reinitialize_breakpoints` variable is set. If all this is true the `reinitialize_breakpoints()` function is called.

Finally `continue_after_breakpoint()` is called and the function returns.

If `address2bpstruct()` call above was not successful, `output_left()` is called to show that an unknown and unexpected breakpoint was hit. The `continue_process()` function is called and the function returns.

22 static void callstack_push_syscall(struct process *proc, int sysnum) – process_event.c

This function simply pushes a `callstack_element` structure onto the array callstack held in the `proc` structure. This structure's `is_syscall` element is set to differentiate this callstack frame from one which represents a library function call. The `proc` structure's member `callstack_depth` is incremented to reflect the callstack's growth.

23 static void callstack_push_symfunc(struct process *proc, struct library_symbol *sym) – process_event.c

As in the `callstack_push_syscall()` function described above, a `callstack_element` structure is pushed onto the array callstack held in the proc structure and the `callstack_depth` element is incremented to reflect this growth.

24 static void callstack_pop(struct process *proc) – process_event.c

This function performs the reverse of the two functions described above. It removes the last structure from the callstack array and decrements the `callstack_depth` element.

25 void enable_all_breakpoints(struct process *proc) – breakpoints.c

This function begins by checking the `breakpoints_enabled` element of the `proc` structure. Only if it is not set the rest of the function continues.

If the architecture is PPC and the option `-L` was **not** used, the function checks if the PLT has been set up by using a ptrace `PTRACE_PEEKTEXT` operation. If not, the function returns at this point.

If `proc->breakpoints` is set the `dict_apply_to_all()` function is called using `enable_bp_cb()` function.[3] This call will set the `proc->breakpoints_enabled`.

[3] This function is a callback that simply calls the function `enable_breakpoint()`.

26 void disable_all_breakpoints(struct process *proc) – breakpoints.c

If `proc->breakpoints_enabled` is set, this function calls `dict_apply_to_all()` with the argument `disable_bp_cb()` as the callback function. It then sets `proc->breakpoints_enabled` to zero and returns.

27 static void disable_bp_cb(void *addr, void *sbp, void *proc) – breakpoints.c

This function is a callback called by `dict_apply_to_all()` and simply calls the function `disable_breakpoint()` (does the reverse of `enable_breakpoint`, copying the saved data from the breakpoint location back over the breakpoint instruction using the ptrace `PTRACE_POKETEXT` interface).

28 void reinitialize_breakpoints(struct process *proc) – breakpoints.c

This function retrieves the list of symbols as a `library_symbol` linked-list structure from the `proc->list_ofsymbols` and iterates over this list, checking each symbol's `need_init` element and calling `insert_breakpoint()` for each symbol for which this is true.

If `need_init` is still set after `insert_breakpoint` an error condition occurs, the error is reported and ltrace exits.

29 void continue_after_breakpoint(struct process *proc, struct breakpoint *sbp) – sysdeps/linux-gnu/trace.c

A check is made to see if the breakpoint is enabled via the `sbp->enabled` flag. If it is then `disable_breakpoint()` is called.

After this, `set_instruction_pointer()`[4] is called to set the instruction pointer to the address of the breakpoint. If the breakpoint is still enabled, then `continue_process()` is called. If not then if the architecture is SPARC or ia64 the `continue_`

[4] This function retrieves the current value of the instruction pointer using the ptrace interface with values of `PTRACE_POKEUSER` and EIP.

process() function is called or if not the ptrace interface is invoked using a PTRACE_SINGLESTEP operation.

30 void open_pid(pid_t pid, int verbose) – proc.c

The trace_pid() function is called on the passed pid, if this fails then the function prints an error message and returns.

The filename for the process is obtained using the pid2name() function and open_program() is called with this filename passed as an argument.

Finally the breakpoints_enabled flag is set in the proc structure returned by open process.

31 static void remove_proc(struct process *proc) – process_event.c

This function removes a process from the linked list of traced processes.

If list_of_processes is equal to proc (i.e., the process was the first in the linked list) then there is a reverse unlink operation where list_of_processes = list_of_processes->next.

If not and the searched-for process is in the middle of the list, then the list is iterated over until the process is found and tmp->next is set to tmp->next->next, simply cutting out the search for process from the linked list.

32 int fork_p(struct process *proc, int sysnum) – sysdeps/linux-gnu/trace.c

This function checks to see if the given sysnum integer refers to a system calls that would cause a child process to be created. It does this by checking the fork_exec_syscalls table using the proc->personality value and an index, *i*, to check each system call in the table sequentially, returning *1* if there is a match.

If the proc->personality value is greater than the size of the table, or should there not be a match, then zero is returned.

33 int exec_p(struct process *proc, int sysnum) – sysdeps/linux-gnu/trace.c

This function checks to see if the given sysnum integer refers to a system calls that would cause another program to be executed. It does this by checking the fork_exec_syscalls table using the proc->personality value and an index, *i*, to check each system call in the table sequentially, returning *1* if there is a match.

If the proc->personality value is greater than the size of the table, or should there not be a match, then zero is returned.

34 void output_line(struct process *proc, char *fmt, ...) – output.c

If the -c option is set, then the function returns immediately. Otherwise the begin_of_line() function[5] is called and the *fmt* argument data is output to the output (can be a file chosen using -o or stderr) using fprintf().

35 void output_left(enum tof type, struct process *proc, char *function_name) – output.c

If the -c option was set, then the function returns immediately. If the current_pid variable is set then the message <unfinished ...> is output and current_pid and current_column are set to zero.

Otherwise current_pid is set to the pid element of the proc structure, and current_depth is set to proc->callstack_depth. The begin_of_line() function is called.

If USER_DEMANGLE is #defined then the function name is output by way of my_demange(), or else it is just output plain.

A variable *func* is assigned by passing the function_name to name2func() if this failed then a loop is iterated four times calling display_arg() many times in succession to display four arguments.

[5] Prints the beginning part of each output line. It prints the process ID, the time passed since the last output line and either the return address of the current function or the instruction pointer.

At the end of the loop it is called a fifth time.

Should the call to `name2func()` succeed, then another loop is iterated but over the number of parameters that the function receives—for each of which the `display_arg()` function is called.

Finally if `func->params_right` is set, `save_register_args()` is called.

36 void output_right(enum tof type, struct process *proc, char *function_name) – output.c

A function structure is allocated via the name2func() function.

If the -c option was set providing the `dict_opt_c` variable is not set it is allocated via a call to `dict_init()`. An `opt_c_struct` structure is allocated by `dict_find_entry()`. If this should fail, then the structure is allocated manually by malloc() and the function name is entered into the dictionary using the `dict_enter()` function.

There are various time calculations and the function returns. If the `current_pid` is set, is not equal to proc->pid and the `current_depth` is not equal to the process' `callstack_depth` then the message `<unfinished>...` is output and `current_pid` is set to zero. If `current_pid` is not equal to the proc structure's pid element then `begin_of_line()` is called and then if `USE_DEMANGLE` is defined the function name is output as part of a resumed message using `fprintf()` via `my_demangle()`. If `USE_DEMANGLE` is not defined then `fprintf()` alone is used to output the message. If *func* is not set then arguments are displayed using `ARGTYPE_UNKNOWN`, otherwise they are displayed using the correct argument type from the `proc` structure.

37 int display_arg(enum tof type, struct process *proc, int arg_num, enum arg_type at) – display_args.c

This function displays one of the arguments, the `arg_num`'th argument to the function the name of which is currently being output to the terminal by the output functions.

It uses a switch case to decide how to display the argument. Void, int, uint, long, ulong, octal char, and address types are displayed using the `fprintf()` stdio function. String and format types are handled by the `display_string`, `display_stringN()` function (sets the `string_maxlength` by calling `gimme_arg()` with the *arg2* variable. It then calls `display_string())` and `display_format()` functions respectively.

Unknown values are handled by the `display_unknown()` function.

38 static int display_unknown(enum tof type, struct process *proc, int arg_num) – display_args.c

The `display_unknown()` function performs a calculation on the argument, retrieved using the `arg_num` variable and uses of the `gimme_arg()` function. Should the value be less than 1,000,000 and greater than –1,000,000 then it is displayed as a decimal integer value; if not, it is interpreted as a pointer.

39 static int display_string(enum tof type, struct process *proc, int arg_num) – display_args.c

The `display_string()` function uses `gimme_arg()` function to retrieve the address of the string to be displayed from the stack. If this fails then the function returns and outputs the string **`NULL`**.

Memory is allocated for the string using `malloc()` and should this fail, the function returns and outputs **`???`** to show that the string was unknown.

The `umovestr()` function is called to copy the string from its address and the length of the string is determined by either the value passed to `-s` or the maximum length of a string (by default infinite). Each character is displayed by the `display_char()` function (outputs the supplied character using `fprintf()`. It converts all the control characters such as \r (carriage return), \n (newline), and EOF (end of file) to printable versions).

Should the string be longer than the imposed maximum string length, then the string "..." is output to show that there was more data to be shown.

The function returns the length of the output.

40 static char *sysname(struct process *proc, int sysnum) – process_event.c

This function retrieves the name of a system call based on its system call number.

It checks the personality element of the proc structure and the sysnum values to check that they fit within the size of the `syscalents[]` array.

If `proc->personality` does not, the `abort()` function is called. If sysnum does not then a string value of `SYS_<sysnum>` is returned.

Provided that both numbers fit within the syscalents array the correct value is obtained using the sysnum variable. A string value of `SYS_<name of systemcall>` is returned.

41 long gimme_arg(enum tof type, struct process *proc, int arg_num) – sysdeps/linux-gnu/*/trace.c

For x86 architecture this function checks if `arg_num` is –1, if so then the value of the EAX register is returned, which is obtained via the ptrace `PTRACE_PEEKUSER` operation.

If type is equal to `LT_TOF_FUNCTION` or `LT_TOF_FUNCTIONR` then the `arg_num`'th argument is returned from the stack via a ptrace `PTRACE_PEEKUSER` operation based on the current stack pointer (from the `proc` structure) and the argument number.

If the type is `LT_TOF_SYSCALL` or `LT_TOF_SYSCALLR` then a register value is returned based on the argument number as so: 0 for EBX, 1 for ECX, 2 for EDX, 3 for ESI, and 4 for EDI.

If the `arg_num` does not match one of the above or the type value does not match either of the above cases, ltrace exits with an error message.

42 static void calc_time_spent(struct process *proc) – process_event.c

This function calculates the time spent in a system call or library function. It retrieves a `callstack_element` structure from the current frame of the process' callstack and calls `gettimeofday()` to obtain the current time and compares the saved time in the `callstack_element` structure to the current time.

This difference is then stored in the `current_diff` variable.

43 void *get_instruction_pointer(struct process *proc) – sysdeps/linux-gnu/*/regs.c

This function retrieves the current value of the instruction pointer using the ptrace interface with values of `PTRACE_PEEKUSER` and EIP.

44 void *get_stack_pointer(struct process *proc) – sysdeps/linux-gnu/*/regs.c

This function retrieves the stack pointer of the traced process by using the ptrace interface with values of `PTRACE_PEEKUSER` and UESP.

45 void *get_return_addr(struct process *proc, void *stack_pointer) – sysdeps/linux-gnu/*/regs.c

This function retrieves the current return address of the current stack frame using the ptrace interface `PTRACE_PEEKTEXT` operation to retrieve the value from the memory pointed to by the current stack pointer.

46 struct dict *dict_init(unsigned int (*key2hash) (void *), int (*key_cmp) (void *, void *)) – dict.c

A dict structure is allocated using `malloc()`, following which the buckets array of this structure is iterated over and each element of the array is set to NULL.

The `key2hash` and `key_cmp` elements of the `dict` structure are set to the representative arguments passed to the function and the function returns.

47 int dict_enter(struct dict *d, void *key, void *value) – dict.c

This function enters a value into the linked list represented by the `dict` structure passed as the first argument.

A hash is calculated by the `key2hash()` function using the key argument to the function and a `dict` structure `new_entry`, which is allocated with `malloc()`. The elements of `new_entry` are set using *key*, *value*, and *hash*.

An index is calculated by rounding the hash value with the size of the `d->bucket` array, and the `new_entry` structure is entered into this array at this index by linking it to the start of the linked list held there.

48 void dict_clear(struct dict *d) – dict.c

This function iterates over both the `d->buckets` array and the linked list held in each `d->buckets` array element. For each linked list element it frees the entry before unlinking it from the list. For each emptied bucket it sets the `d->bucket` element to NULL.

49 void *dict_find_entry(struct dict *d, void *key) – dict.c

A hash is created using the `d->key2hash` function pointer and the passed key argument variable.

This hash is then used to index into the `d->buckets` array as a `dict_entry` structure. The linked listed held in this element of the array is iterated over comparing the calculated hash value to the hash value held in each element of the linked list.

Should the hash values match, a comparison is made between the key argument and the key element of this linked list. If this comparison should prove true the function returns the entry. Otherwise the function returns NULL if no matches are ultimately found.

50 void dict_apply_to_all(struct dict *d, void (*func) (void *key, void *value, void *data), void *data) – dict.c

This function iterates over all the elements in the `d->buckets` array, and iterates over the linked list held in each element of said array.

For each element of each linked list the passed func function pointer is called using the key, value and data elements of the supplied dict structure *d*.

51 unsigned int dict_key2hash_string(void *key) – dict.c

This function creates a hash value from a character string passed as the void pointer *key*.

The key is first cast to a character pointer and for each character in this string the following is carried out:

- The integer *total* is incremented by the current value of total XORd by value of the character shifted left by the value shift, which starts out as zero, and is incremented by five for each iteration.
- Should the shift pass the value of 24, it is reduced to zero.

After processing each character in the supplied string the function returns the value held in the variable *total* as the final hash value.

52 dict_key_* helper functions – dict.c

Ltrace have many simple function to help in the key comparisions:

- `int dict_key_cmp_string(void *key1, void *key2) -- dict.c`

  A very simple function that returns the result of a call to `strcmp()` using the two supplied pointer values.

- `unsigned int dict_key2hash_int(void *key) -- dict.c`

  This is a very simple function that returns the supplied pointer value cast to an unsigned int type.

- `int dict_key_cmp_int(void *key1, void *key2) -- dict.c`

  This is a very simple function that returns the mathematical difference of *key2* from *key1*.

Evaluating effects of cache memory compression on embedded systems

Anderson Farias Briglia
Nokia Institute of Technology
anderson.briglia@indt.org.br

Allan Bezerra
Nokia Institute of Technology
allan.bezerra@nokia.com

Leonid Moiseichuk
Nokia Multimedia, OSSO
leonid.moiseichuk@nokia.com

Nitin Gupta
VMware Inc.
ngupta@vmware.com

Abstract

Cache memory compression (or compressed caching) was originally developed for desktop and server platforms, but has also attracted interest on embedded systems where memory is generally a scarce resource, and hardware changes bring more costs and energy consumption. Cache memory compression brings a considerable advantage in input-output-intensive applications by means of using a virtually larger cache for the local file system through compression algorithms. As a result, it increases the probability of fetching the necessary data in RAM itself, avoiding the need to make low calls to local storage. This work evaluates an Open Source implementation of the cache memory compression applied to Linux on an embedded platform, dealing with the unavoidable processor and memory resource limitations as well as with existing architectural differences.

We will describe the Compressed Cache (CCache) design, compression algorithm used, memory behavior tests, performance and power consumption overhead, and CCache tuning for embedded Linux.

1 Introduction

Compressed caching is the introduction of a new level into the virtual memory hierarchy. Specifically, RAM is used to store both an uncompressed cache of pages in their 'natural' encoding, and a compressed cache of pages in some compressed format. By using RAM to store some number of compressed pages, the effective size of RAM is increased, and so the number of page faults that must be handled by very slow hard disks is decreased. Our aim is to improve system performance. When that is not possible, our goal is to introduce no (or minimal) overhead when compressed caching is enabled in the system.

Experimental data show that not only can we improve data input and output rates, but also that the system behavior can be improved, especially in memory-critical cases leading, for example, to such improvements as postponing the out-of-memory activities altogether. Taking advantage of the kernel swap system, this implementation adds a virtual swap area (as a dynamically sized portion of the main memory) to store the compressed pages. Using a dictionary-based compression algorithm, page cache (file-system) pages and anonymous pages are compressed and spread into variable-sized memory chunks. With this approach, the fragmentation can be reduced to almost zero whilst achieving a fast page recovery process. The size of Compressed Cache can be adjusted separately for Page Cache and Anonymous pages on the fly, using `procfs` entries, giving more flexibility to tune system to required use cases.

2 Compressed Caching

2.1 Linux Virtual Memory Overview

Physical pages are the basic unit of memory management [8] and the MMU is the hardware that translates virtual pages addresses into physical pages address and vice-versa. This compressed caching implementation, CCache [3], adds some new flags to help with compressed pages identification and uses the same lists used by the PFRA (Page Frame Reclaiming Algorithm). When the system is under a low memory condition, it evicts pages from memory. It uses Least Recently Used (LRU) criteria to determine order in which

to evict pages. It maintains two LRU lists—active and inactive LRU lists. These lists may contain both page-cache (file-backed) and swap-cache (anonymous) pages. When under memory pressure, pages in inactive list are freed as:

- Swap-cache pages are written out to swap disks using `swapper_space writepage()` (`swap_writepage()`).
- Dirty page-cache pages are flushed to filesystem disks using filesystem specific `writepage()`.
- Clean page-cache pages are simply freed.

2.1.1 About Swap Cache

This is the cache for anonymous pages. All swap cache pages are part of a single `swapper_space`. A single radix tree maintains all pages in the swap cache. `swp_entry_t` is used as a key to locate the corresponding pages in memory. This value identifies the location in swap device reserved for this page.

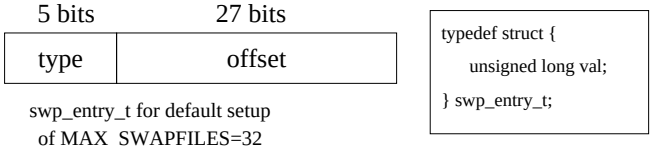


Figure 1: Fields in `swp_entry_t`

In Figure 1, **'type'** identifies things we can swap to.

2.1.2 About Page Cache

This is the cache for file-system pages. Like swap cache, this also uses radix-tree to keep track of file pages. Here, the offset in file is used as the search key. Each open file has a separate radix-tree. For pages present in memory, the corresponding radix-node points to `struct page` for the memory page containing file data at that offset.

2.2 Compressed Cache Overview

For compressed cache to be effective, it needs to store both swap-cache and page-cache (clean and dirty) pages. So, a way is needed to transparently (i.e., no changes required for user applications) take these pages in and out of compressed cache.

This implementation handles anonymous pages and page-cache (filesystem) pages differently, due to the way they are handled by the kernel:

- For anonymous pages, we create a **virtual swap**. This is a memory-resident area of memory where we store compressed anonymous pages. The swap-out path then treats this as yet another swap device (with highest priority), and hence only minimal changes were required for this kernel part. The size of this swap can be dynamically adjusted using provided `proc` nodes.
- For page-cache pages, we make a corresponding page-cache entry point to the location in the compressed area instead of the original page. So when a page is again accessed, we decompress the page and make the page-cache entry point back to this page. We did not use the 'virtual swap' approach here since these (file-system) pages are never 'swapped out.' They are either flushed to file-system disk (for dirty pages) or simply freed (for clean pages).

In both cases, the actual compressed page is stored as series of variable sized 'chunks' in a specially managed part of memory which is designed to have minimum fragmentation in storing these variable-sized areas with quick storage/retrieval operations. All kinds of pages share the common compressed area.

The compressed area begins as few memory pages. As more pages are compressed, the compressed area inflates (up to a maximum size which can be set using `procfs` interface) and when requests for these compressed pages arrive, these are decompressed, and corresponding memory 'chunks' are put back onto the *free-list*.

2.3 Implementation Design

When a page is to be compressed, the radix node pointing to the page is changed to point to the **chunk_head**—this in turn points to first of the **chunks** for the compressed page and all the chunks are also linked. This `chunk_head` structure contains all the information

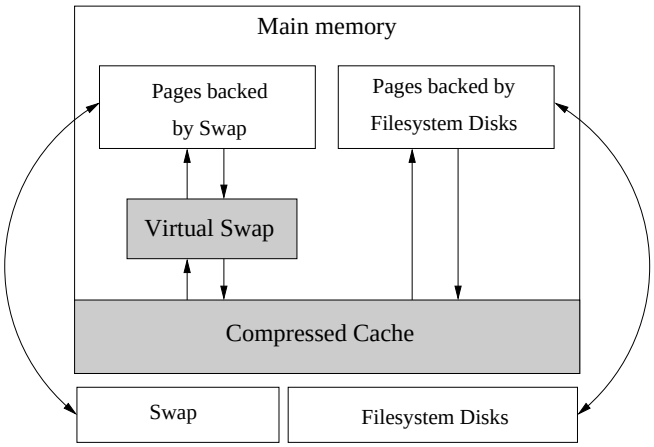


Figure 2: Memory hierarchy with Compressed Caching

required to correctly locate and decompress the page (compressed page size, compression algorithm used, location of first of chunks, etc.).

When the compressed page is accessed/required later, page-cache/swap-cache (radix) lookup is done. If we get a `chunk_head` structure instead of a page structure on lookup, we know this page was compressed. Since `chunk_head` contains a pointer to first of the chunks for this page and all chunks are linked, we can easily retrieve the compressed version of the page. Then, using the information in the `chunk_head` structure, we decompress the page and make the corresponding radix-node points back to this newly decompressed page.

2.3.1 Compressed Storage

The basic idea is to store compressed pages in variable-sized memory blocks (called *chunks*). A compressed page can be stored in several of these chunks. Memory space for chunks is obtained by allocating 0-order pages at a time and managing this space using chunks. All the chunks are always linked as a doubly linked list called the *master chunk list*. Related chunks are also linked as a singly linked list using the related-chunk list; e.g., all free chunks are linked together, and all chunks belonging to the same compressed page are linked together. Thus all chunks are linked using master chunk list and related chunks are also linked using one of related-chunk list (e.g. free list, chunks belonging to same compressed page).

Note that:

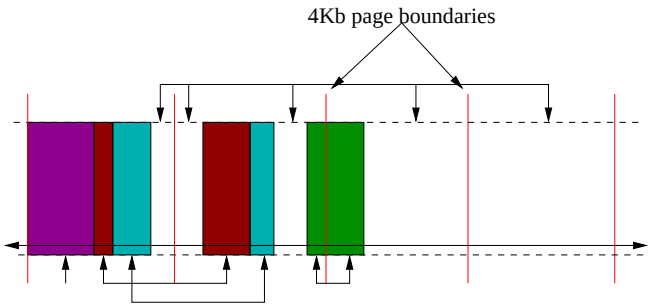


Figure 3: A sample of compressed storage view highlighting 'chunked' storage. Identically colored blocks belong to the same compressed page, and white is free space. An arrow indicates related chunks linked together as a singly linked list. A long horizontal line across chunks shows that these chunks are also linked together as a doubly lifnked list in addition to whatever other lists they might belong to.

- A chunk cannot cross page boundaries, as is shown for the 'green' compressed page. A chunk is split unconditionally at page boundaries. Thus, the maximum chunk size is `PAGE_SIZE`.
- This structure will reduce fragmentation to a minimum, as all the variable, free-space blocks are being tracked.
- When compressed pages are taken out, corresponding chunks are added to the free-list and physically adjacent free **chunks are merged together** (while making sure chunks do not cross page boundaries). If the final merged chunk spans an entire page, the page is released.

So, the compressed storage begins as a single chunk of size `PAGE_SIZE` and the free-list contains this single chunk.

An LRU-*list* is also maintained which contains these chunks in the order in which they are added (i.e., the 'oldest' chunk is at the tail).

2.3.2 Page Insert and Delete

Page Insert: The uncompressed page is first compressed in a buffer page. Then a number of free chunks are taken from free list (or a new page is allocated to get a new chunk) according to the size of the compressed page. These chunks are linked together as a singly

linked related-list. The remaining space from the last chunk is added back to the free list, and merged with adjacent free chunks, if present. The entry in the page-cache radix tree is now made to point to the `chunk_head` allocated for this compressed page. Pages that are not compressible (size increases on compression), are never added to CCache. The usual reclaim path applies to them.

Page Delete: When a page is looked up in memory, we normally get a `struct page` corresponding to the actual physical page. But if the page was compressed, we instead get a `struct chunk_head` rather than a `struct page` at the corresponding node (identified by the `PG_compressed` flag set), which gives a link to first chunk. Since all related chunks are linked together, compressed data is collected from all these chunks in a separate page, and then decompression is done. The freed chunks are merged with adjacent free chunks, if present, and then added to the free list.

2.4 Compression Methods

In the fundamental work [5], a number of domain-specific considerations are discussed for compression algorithms:

1. Compression must be lossless, i.e., one-to-one mapping from compressed and decompressed forms, so exactly the same, original data must be restored.

2. Compression must be in memory, so no kind of external memory can be used. Additionally, the following properties of typical memory data can be exploited to achieve good compression:

   - it is word or double-word aligned for faster handling by the processor;
   - it contains a large number of integers with a limited range of values;
   - it contains pointers (usually with integer size), mostly to the same memory region, so the majority of information is stored in the lower bits;
   - it contains many regularity sequences, often zeros.

3. Compression must incur a low overhead. Speed matters for both compression and decompression. A compression algorithm can also be asymmetric since typically decompression is required for many read operations, therefore making it important to have as cheap a decompression as possible. Small space overhead is also required since this overhead memory has to be allocated when the system is already running low on memory.

4. Compressible data, on average, should incur a 50% size reduction. If a page can not be compressed up to 50%, it increases the complexity of the procedure for handling compressed pages.

Thus, general purpose compression algorithms may not be good for our domain-specific compression requirements, due to high overhead. In this case we can choose low overhead compression, which takes into account the majority of in-memory data regularities and produces a sufficient compression ratio. We can also balance between compressibility and overhead by using several compression algorithms $\{A1..AN\}$ sequentially. Assuming that probabilities to compress a page are $\{P1..PN\}$ and compression times are $\{C1..CN\}$, the average compression time can be estimated as

$$C = \sum C_1 * P_1 \tag{1}$$

Note that since non-compressible pages can exist, we can determine that

$$\sum Pi < 1.0 \tag{2}$$

Thus, the best result from the speed versus compressibility point of view will be obtained by minimizing C time at compile- or run-time. The simplest way is to apply first the fastest algorithm, then the second fastest, and so on, leaving the slowest as last. Typically this scheme will work pretty well if $C1 \ll CN$; nevertheless, any runtime adoption can be used. Originally the following compression methods were used for cache memory compression [5]:

- WK in-memory compression family of algorithms as developed by Paul Wilson and Scott F. Kaplan. These algorithms are based on the assumption that

the target system has 32-bit word size, 22 bits of which match exactly, and 10 bits differ in values which will be looked for. Due to this assumption, this family of algorithms is not suitable for 64-bit systems, for which another lookup approach should be used. The methods perform close to real-life usage:

  - WK4x4 is the variant of WK compression that achieves the highest (tightest) compression by itself by using a 4x4 set-associative dictionary of recently seen words. The implementation of a recency-based compression scheme that operates on machine-word-sized tokens.
  - WKdm compresses nearly as tightly as WK4x4, but is much faster because of the simple, direct-mapped dictionary of recently seen words.

- miniLZO is a lightweight subset of the very fast Lempel-Ziv (LZO) implementation by Markus Oberhumer [9], [15]. Key moments can be highlighted from the description and make this method very suitable for our purpose:

  - Compression is pretty fast, requires 64KB of memory.
  - Decompression is simple and fast, requires no memory.
  - Allows you to dial up extra compression at a speed cost in the compressor. The speed of the decompressor is not reduced.
  - Includes compression levels for generating pre-compressed data, which achieves a quite competitive compression ratio.
  - There is also a compression level which needs only 8 KB for compression.
  - Algorithm is thread-safe.
  - Algorithm is lossless.
  - LZO supports overlapping compression and in-place decompression.
  - Expected speed of LZO1X-1 is about 4–5 times faster than the fastest zlib compression level.

3 Experimental Results

This section will describe how the CCache was tested on an OMAP platform device—the Nokia Internet Tablet N800 [6]. The main goal of the tests is to evaluate the characteristics of the Linux kernel and overall system behavior when CCache is added in the system.

As said before, this CCache implementation can compress two types of memory pages: anonymous pages and page-cache pages. The last ones never go to swap; once they are mapped on a block device file and written to the disk, they can be freed. But anonymous pages are not mapped on disk and should go to swap when the system needs to evict some pages from memory.

We have tests with and without a real swap partition, using a MMC card. Our tests intend to evaluate CCache against a real swap device. So, we decided to use just anonymous pages compression. This way we can better compare CCache performance against a system with real swap. For the comparison, we measured the following quantities:

- How many pages were maintained in memory (CCache) and did not go to swap (avoiding I/O overhead).
- Changes in power requirements.
- Comparison of different compression algorithms: compress and decompress times, compression ratio.

3.1 Test Suite and Methodology

We used a mobile device with embedded Linux as testing platform. The Nokia Internet Tablet N800 has a 330Mhz ARM1136 processor from Texas Instruments (OMAP 2420), 128MB of RAM, and 256MB of flash memory used to store the OS and applications. The OMAP2420 includes an integrated ARM1136 processor (330 MHz), a TI TMS320C55x DSP (220 MHz), 2D/3D graphics accelerator, imaging and video accelerator, high-performance system interconnects, and industry-standard peripherals [4]. Two SD/MMC slots, one internal and another external, can be used to store audio/video files, pictures, documents, etc.

As we can see on device's specification, it is focused on multimedia usage. As we know, multimedia applications have high processor and memory usage rates. Our tests intend to use memory intensively, hence we can measure the CCache impact on system performance. The tests were divided into two groups: tests with real applications and tests using synthetic benchmarks. Tests with real applications tried to simulate a high memory consumption scenario with a lot of applications being executed and with some I/O operations to measure the memory and power consumption behavior when CCache is running. Synthetic tests were used to evaluate the CCache performance, once they provided a easy way to measure the time spent on the tests.

Tests with real applications consist of:

- Running 8 or 9 Opera browsers and load some pages through a wireless Internet connection.
- Playing a 7.5MB video on Media player.
- Opening a PDF document.

We used an application called **xautomation** [12] to interact with the X system through bash scripts from the command line. Xautomation controls the interface, allowing mouse movements, mouse right and left clicks, key up and down, etc. Reading the `/proc/meminfo` file, we have the memory consumption, and some graphics related to the memory consumption can be plotted. They will be shown and commented on in the following sections.

The tests using synthetic benchmarks were executed using MemTest [11]. MemTest is a group of small tests to evaluate the stability and consistency of the Linux memory management system. It contains several tests, but we used just one, since our main goal is to measure the CCache performance: **fillmem**. This test intends to test the system memory allocation. It is useful to verify the virtual memory system against the memory allocation operation, pagination, and swap usage. It has one parameter which defines the size of memory allocated by itself.

All the tests, using real applications or synthetic benchmarking, were applied using a pre-configured scenarios, depending on what would be measured. Basically we have scenarios with different RAM memory sizes, with or without a *real* swap partition, and with or without CCache added to the system.

Memory behavior tests evaluate the memory consumption against time and the OOM killer interaction on the scenarios discussed before. Performance tests used MemTest [11] to measure the total time of the `fillmem` execution. Power consumption tests evaluate the impact of CCache usage on power consumption, since it is crucial on mobile devices with embedded systems.

3.1.1 Tuning for Embedded Linux

One of the most important constraints in embedded systems is the storage space available. Therefore implementing mechanisms to save some space is crucial to improve the embedded OS.

The N800's OS, also known as Internet Tablet OS 2007 (IT OS 2007), is based on the linux-omap [10] kernel and has some features customized to this device. One of these, the most relevant in this case, is the swap system behavior. The Linux kernel virtual memory subsystem (VM) operation can be tuned using the files included at `/proc/sys/vm` directory. There we can configure OOM-killer parameters, swap parameters, writeout of dirty data to disk, etc. On this case, just swap-related parameters were modified, since the evaluation must be as close as possible to reality—in the other words, the default system configuration.

The swap system behavior on the device's kernel is configured as if a swap partition were not present in the system. We configured two parameters to make the test execution feasible under low-memory scenarios: `/proc/sys/vm/swappiness` and `/proc/sys/vm/min_free_kbytes`.

- **swappiness** [7] is a parameter which sets the kernel's balance between reclaiming pages from the page cache and swapping process memory. The default value is 60.
- **min_free_kbytes** [7] is used to force the Linux VM to keep a minimum number of kilobytes free.

If the user wants the kernel to swap out more pages, which in effect means more caching, the `swappiness`

parameter must be increased. The `min_free_kbytes` controls when the kernel will start to force pages to be freed, before out-of-memory conditions occur.

The default values for `swappiness` and `min_free_kbytes` are **0** and **1280**, respectively. During our tests, we had to modify these values since the CCache uses the default swap system, and with the default values, the CCache would not be used as expected. Another fact is that we added a real swap partition, and the `swappiness` must be adjusted to support this configuration. Increasing the `swappiness` to its default value, 60, we noticed that more pages were swapped and the differences between the CCache usage and a real swap partition could be measured.

During our tests the available memory was consumed so quickly that CCache could not act. The OOM killer was called, and the test was killed. To give more time to CCache, the `min_free_kbytes` was increased. Another motivation is that pages taken by CCache are marked as pinned and never leave memory; this can contribute to anticipating the OOM killer call. It happens because the CCache pages are taken out from the LRU list and due to this, the PFRA (Page Frame Reclaiming Algorithm) cannot select such pages to be freed.

Other parameters that must be adjusted are the `max_anon_cc_size` and `max_fs_backed_cc_size`. These parameters set the max size for anonymous pages and the max size for page-cache pages, respectively. The CCache will increase until the max size is reached. Note that those parameters are set in number of pages (4KB) and the pages are used to store the chunks lists and the metadata needed to manage those chunks. These parameters are exported by CCache via `/proc/sys/vm/max_anon_cc_size` and `/proc/sys/vm/max_fs_backed_cc_size`. We have one limitation here: `max_anon_cc_size` and `max_fs_backed_cc_size` can be configured only one time; dynamic re-sizing is not permitted yet.

As previously mentioned, the tests covered the anonymous pages compression; therefore only `max_anon_cc_size` was used. After the initialization, the IT OS 2007 has about 50M of free memory available, from a total of 128M. After making some empirical tests, we decided to set `max_anon_cc_size` to about 10% of device's total free memory (5 MB), or in other words, 1280 pages of 4 KB each. Since CCache pages are pinned in memory and adaptive resizing is not yet implemented, we chose to have only a small percentage of memory assigned to ccache (10%). With a higher percentage, we might end-up calling the OOM killer too soon.

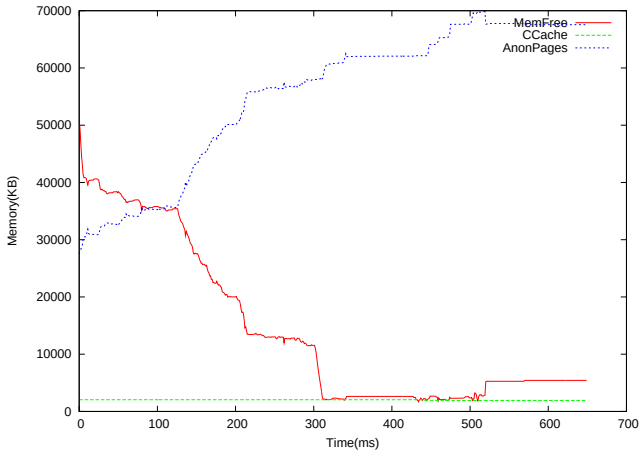


Figure 4: CCache x time: max_anon_cc_size = 1024, Mem = 128M

3.2 Memory Behavior Tests

The main goal of these tests is to see what is happening with the memory when CCache is compressing and decompressing pages. To execute the tests we used real application scenarios, using applications provided by the default installation of the N800's distribution.

Using a bash script with XAutomation [12], we can interact with the X server and load some applications while another program collects the memory variables present in `/proc/meminfo` file. Figure 4 shows the memory consumption against time on a kernel with CCache and `max_anon_cc_size` set to 1024 pages (4MB).

As we can see in Figure 4, the CCache consumption was very low once the `swappiness` parameter was configured with default value of 1. Therefore more pages are cached in memory instead of going to swap, even if the available free memory is low. Actually this behavior is expected since the system doesn't have swap for default. On Figure 5, we limited the memory to 100M at boot, which caused increased memory pressure on the system. The figure shows that the OOM killer was called at 600 ms, and CCache did not have time to react.

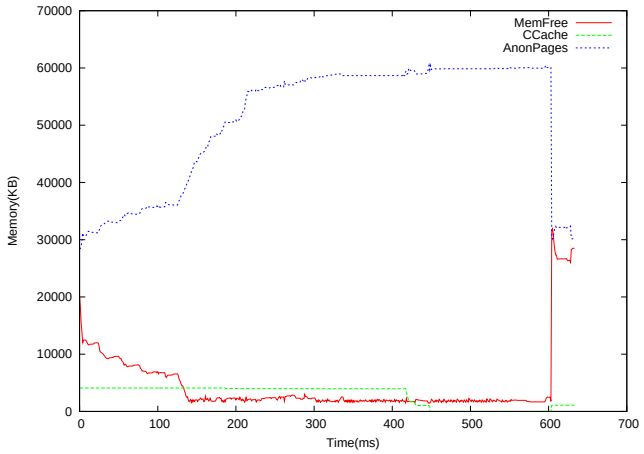


Figure 5: CCache x time: max_anon_cc_size = 1024, Mem = 100M

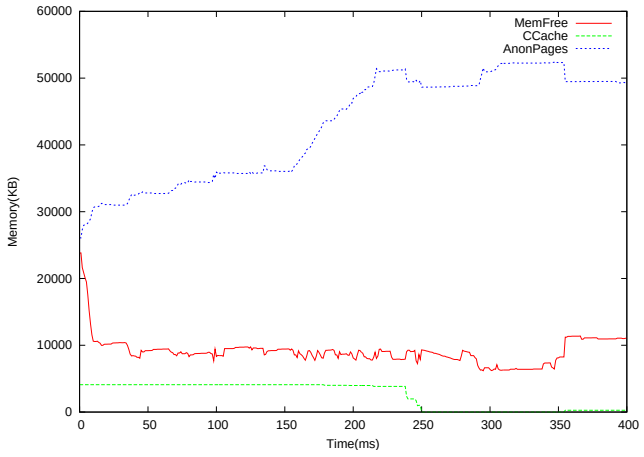


Figure 6: CCache x time: swappiness = 60 and min_free_kbytes = 3072

Using the same test but with `swappiness` and `min_free_kbytes` configured with 60 and 3072 respectively, the OOM killer is not called any more, and we have an increased usage of CCache. See Figure 6.

Figure 6 shows that the CCache size is not enough for the memory load of the test. But we have a problem here: if the CCache size is increased, more pages are taken off the LRU list and cannot be freed. The best solution here is implementing the adaptive [1, 13] CCache resizing. We will cover a bit more about adaptive CCache in Section 4.

In the tests, the new page size was collected for each page that was compressed. With this, we can have measurements using the most common ranges of compressed page size. Figure 7 illustrates the Compression Page Size Distribution in five ranges, pages sized between: 0K–0.5K, 0.5K–1K, 1K–2K, 2K–3K and 3K–4K. About 42% of pages have sizes between 1K and 2K after compression.

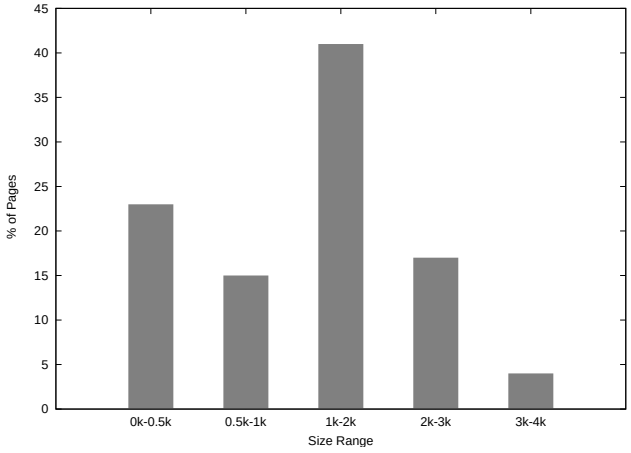


Figure 7: Compression size distribution

These results indicate that we have a general system gain with CCache. Since most of pages sizes are between 1K–2K, and we adopted a chunk-based approach with no fragmentation. In general, we have an increase of 100% of 'system available memory.' It is an important CCache advantage: applications that before, could not be executed on the system, now have more 'visible available memory' and can be executed, allowing an embedded system to support applications that it would otherwise be unable to run. It is important to remember that on CCache, compressed pages with size more than 4KB are discarded.

3.3 Performance Tests

Performance tests aim to analyze the CCache overhead: compression overhead, chunks lists handling overhead, and page recovery overhead. With these tests we expect to prove that CCache, even with all those overheads, is faster than using a swap partition on a block device.

Only anonymous pages are being considered here. As explained in Section 2.2, there are some steps when a page is compressed. Table 1 shows the results when running the fillmem allocating 70MB of memory (`swappiness = 60, min_free_kbytes = 1024KB, max_cc_anon_size = 1280 pages`):

The test using WK4x4 triggered the OOM killer and could not be completed. But taking a look at the other

Test	No-CCache	CCache WKdm	CCache WK4x4	CCache LZO
Time(s)	14.68	4.64	–	4.25

Table 1: fillmem(70): CCache time x Real Swap time

values, we can conclude that the swap operation using CCache is about **3 times** faster than using a swap partition on an MMC card. But how about the OOM-killer?

As discussed before, CCache does not have dynamic resizing. It implies that for some use cases, CCache has no benefits and the memory allocated to the compressed data becomes prejudicial to the system. Since we do not have the adaptativity feature implemented, it is very important to tune CCache according the use case.

3.3.1 Compression Algorithms Comparison

As we wrote above, it is essential to minimize overhead by using an optimal compression algorithm. To compare a number of algorithms, we performed tests on the target N800 device. In addition to those algorithms used in CCache (WKdm, WK4x4, zlib, and miniLZO), we tested methods used in the Linux kernel (rubin, rtime) and LZO compression.

Speed results are shown below relative to `memcpy` function speed, so 2.4 means something works 2.4 times slower than `memcpy`. Reported data is the size of data produced by compiler (`.data` and `.bss` `segments`). The test pattern contained 1000 4K-pages of an ARM ELF file which is the basic test data for CCache. Input data was compressed page-per-page (4 KB slices).

Name	ARM11 code size (bytes)	ARM11 data size (bytes)	Comp time (relative)	Comp ratio (%)	Decomp time (relative)	Speed asymm.
Wkdm	3120	16	2.3	89	1.4	1.8
Wk4x4	4300	4	3.5	87	2.6	0.9
miniLZO	1780	131076	5.6	73	1.6	3.5
zlib	49244	716	73.5	58	5.6	13.1
rubin_mips	180	4	152	98	37	4.2
dyn_rubin	180	4	87.8	92	94.4	0.9
rtime	568	4	7	97	1	7
lzo1	2572	0	8.8	76	1.6	5.5
lzo2	4596	0	62.8	62	2	31.4

Table 2: Compression Algorithms Comparison

From these tests we can draw the following conclusions for the target system:

1. zlib is a very slow method, but it provides the best compression;
2. using WK4x4 makes no practical sense because the number of pages compressed for 50% or better is the same as for WKdm, but in terms of speed, WK4x4 is about 2 times slower;
3. lzo2 has good compression and very good decompression speed, so it will be used to replace zlib everywhere.
4. by using WKdm and miniLZO sequentially, we can obtain good compression levels with small overhead.

3.4 Power Consumption Tests

In order to evaluate the power consumption using CCache, we replaced the device's battery with an Agilent direct current power supply. This Agilent equipment shows the total current in real time and with this, the power consumption was calculated using an output voltage of 4V.

Figure 8 shows the power consumption for the interactive user test used and described in Section 3.2. Steps 1 to 5 can be discarded in an analysis of power consumption since in these steps CCache is active, but it is not called. The important steps are 5 to 8. In these steps we have a stress memory situation and CCache is working. As we can see, we have a gain of about 3% on average when compression is used. It can be explained for smaller I/O data rates. Is important to take a look at the 'Video Play' step: this kind of application needs more energy to process the video stream. These results show that in this situation we have a good gain when CCache is working. It is important to note that some variation is accepted since all interactive tests results are use-case dependent.

4 Related Work

The first implementation of compressed cache was done by Douglis [2] in 1993 and results were not conclusive. He achieved speed up for some applications, and slowdowns for others. Later, in 1999 Kaplan [5] pointed out that the machine Douglis used had a difference between the processor speed and the access times on hard disk

	Tests Step	Power Consumption for Max Corrent (W) Voltage used was 4V	
		W/O CC	With CC
Browser Pages	1	2,332	2,328
	2	2,136	2,184
	3	2,348	2,360
	4	2,392	2,384
	5	2,448	2,372
	6	2,492	2,440
	7	2,484	2,436
	8	2,572	2,444
Video Play	9	2,880	2,804
PDF File Open	10	2,528	2,480
File Manager Filesystem Scan	11	2,356	2,316

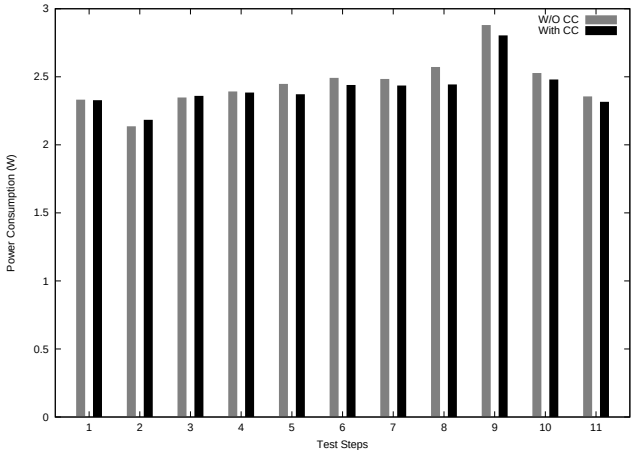


Figure 8: Power Consumption tests using CCache

that was much smaller than encountered nowadays. Kaplan also proposed a new adaptive scheme, since that used by Douglis was not conclusive about the applications' performance.

Following the same scheme, Rodrigo Castro [1] implemented and evaluated compressed cache and compressed swap using the 2.4.x Linux kernel. He proposed a new approach to reduce fragmentation of the compressed pages, based on contiguous memory allocated areas—cells. He also proposed a new adaptability policy that adjusts the compressed cache size on the fly. Rodrigo's adaptive compressed cache approach is based on a tight compressed cache, without allocation of superfluous memory. The cells used by compressed cache are released to the system as soon as they are no longer needed. The compressed cache starts with a minimum size and as soon as the VM system starts to evict pages, the compressed cache increases its memory usage in order to store them.

All those implementations are focused on desktop or server platforms. One implementation which is focused on embedded devices is CRAMES [14]—Compressed RAM for embedded systems. CRAMES was implemented as a loadable module for the Linux kernel and evaluated on a battery-powered embedded system. CRAMES supports in-RAM compressed filesystems of any type and uses a chunks approach (not the same described in this paper) to store compressed pages.

The compressed cache implementation evaluated in this paper still does not have the adaptive feature implemented. From previous work, CCache uses the same compression algorithms used by Rodrigo's implementation, but the storage of compressed pages is quite different. The chunks approach reduces the fragmentation close to zero and speeds up the page recovery. CCache is the first Open Source compressed cache implementation for 2.6.x Linux kernel and it is under development, both for desktop/server platforms and for embedded devices, as presented in this paper.

5 Conclusions and Future Work

Storing memory pages as compressed data decreases the number of access attempts to storage devices such as, for example, hard disks, which typically are accessed much slower than the main memory of a system. As such, we can observe more benefits when the difference between the access time to the main memory and the storage device is considerable. This characteristic is not typical for embedded Linux systems, and the benefits of storing pages as compressed data are much smaller than on an x86 architecture. Storage devices in the embedded Linux systems are typically flash memory, for instance MMC cards, and as we know, access times for these devices are much faster than access times for a hard disk. It allows us to come to the conclusion that wide use of CCache is not justified in the embedded systems.

On the other hand, embedded systems have limitations in the available memory. Thus, the experimental tests results show that CCache can improve not only the input and output performance but the system behavior in general by improving memory management like swapping, allocating big pieces of memory, or out-of-memory handling. Being that as it may, it is also common knowledge that this scenario has been changing along the development of embedded Linux. The best benefit of this CCache implementation is that developers can adjust it to its system characteristics. Implementing a software-based solution to handle the memory limitations is much

better than hardware changes, which can increase the market price of a product.

The power tests show that CCache makes a positive impact on power consumption, but for the N800 architecture the overall benefit is minor due to the high level of system integration which is based on the OMAP 2420.

Finally, we can make some important points which can improve this CCache implementation:

- use only fast compression methods like WKdm and minilzo, maybe with lzo2;
- make performance and memory optimization for compression take into account CPU and platform hardware-accelerating features;
- enable compression according to instant memory load: fast methods when memory consumption is moderated, and slow when high memory consumption is reached.
- compress/decompress pages that go to/from a real swap partition.
- adaptive size of compressed cache: detection of a memory consumption pattern and predict the CCache size to support this load.

6 Acknowledgements

First we have to thank Nitin Gupta for designing and implementing CCache on Linux 2.6.x kernels. We would like to thank Ilias Biris for the paper's revision and suggestions, Mauricio Lin for helping on the Linux VM subsystem study, and Nokia Institute of Technology in partnership with Nokia for providing the testing devices and support to work on this project.

References

[1] Rodrigo Souza de Castro, Alair Pereira Lago, and Dilma da Silva. Adaptive compressed caching: Design and implementation. 2003. http://linuxcompressed.sourceforge.net/docs/files/paper.pdf.

[2] Douglis F. The compression cache: Using on-line compression to extend physical memory. 1993. http://www.lst.inf.ethz.ch/research/publications/publications/USENIX_2005/USENIX_2005.pdf.

[3] Nitin Gupta. Compressed caching for linux project's site, 2006. http://linuxcompressed.sourceforge.net/.

[4] Texas Instruments, 2006. http://focus.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?contentId=4671&navigationId=11990&templateId=6123.

[5] S. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation*. PhD thesis, University of Texas at Austin, 1999.

[6] LinuxDevices.com, 2007. http://www.linuxdevices.com/Articles/AT9561669149.html.

[7] linux.inet.hr, 2007. http://linux.inet.hr/proc_sys_vm_hierarchy.html.

[8] Robert Love. *Linux Kernel Development*. Novell Press, 2nd edition, 2005. ISBN 0-672-32720-1.

[9] Markus Franz Xaver Johannes Oberhumer, 2005. http://www.oberhumer.com/opensource/lzo/lzodoc.php.

[10] Linux omap mailing list, 2007. http://linux.omap.com/mailman/listinfo/linux-omap-open-source.

[11] Juan Quintela, 2006. http://carpanta.dc.fi.udc.es/~quintela/memtest/.

[12] Steve Slaven, 2006. http://hoopajoo.net/projects/xautomation.html.

[13] Irina Chihaia Tuduce and Thomas Gross. Adaptive main memory compression. 2005. http://www.lst.inf.ethz.ch/research/publications/publications/USENIX_2005/USENIX_2005.pdf.

[14] Lei Yang, Robert P. Dick, Haris Lekatsas, and Srimat Chakradhar. Crames: Compressed ram for

embedded systems. 2005. http://ziyang.eecs.northwestern.edu/~dickrp/publications/yang05sep.pdf.

[15] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

ACPI in Linux® – Myths vs. Reality

Len Brown
Intel Open Source Technology Center
len.brown@intel.com

Abstract

Major Linux distributors have been shipping ACPI in Linux for several years, yet mis-perceptions about ACPI persist in the Linux community. This paper addresses the most common myths about ACPI in Linux.

1 Myth: There is no benefit to enabling ACPI on my notebook, desktop, or server.

When users boot in ACPI-mode instead of legacy-mode, the first thing they notice is that the power button is now under software control. In legacy-mode, it is a physical switch which immediately removes power. In ACPI mode, the button interrupts the OS, which can shutdown gracefully. Indeed, ACPI standardizes the power, sleep, and lid buttons and the OS can map them to whatever actions it likes.

In addition to standardizing power button events, ACPI also standardizes how the OS invokes software controlled power-off. So a software-initiated power off removes power from the system after Linux has shutdown, while on many systems in legacy-mode, the operator must manually remove power.

Users notice improved battery life when running in ACPI-mode. On today's notebooks, a key contributor to improved battery life is processor power management. When the processor is idle, the Linux idle loop takes advantage of ACPI CPU idle power states (C-states) to save power. When the processor is partially idle, the Linux cpufreq sub-system takes advantage of ACPI processor performance states (P-states) to run the processor at reduced frequency and voltage to save power.

Users may notice other differences depending on the platform and the GUI they use, such as battery capacity alarms, thermal control, or the addition of or changes in the the ability to invoke suspend-to-disk or suspend-to-RAM, etc.

Users with an Intel® processor supporting Hyper-Threaded Technology (HT) will notice that HT is enabled in ACPI-mode, and not available in legacy-mode.

Many systems today are multi-processor and thus use an IO-APIC to direct multiple interrupt sources to multiple processors. However, they often do not include legacy MPS (Multi-Processor Specification) support. Thus, ACPI is the only way to configure the IO-APIC on these systems, and they'll run in XT-PIC mode when booted without ACPI.

But to look at ACPI-mode vs. legacy-mode a bit deeper, it is necessary to look at the specifications that ACPI replaces. The most important one is Advanced Power Management [APM], but ACPI also obsoletes the Multi-Processor Specification [MPS] and the PCI IRQ Routing Specification [PIRQ].

1.1 ACPI vs. Advanced Power Management (APM)

APM and ACPI are mutually exclusive. While a flexible OS can include support for both, the OS can not enable both on the same platform at the same time.

APM 1.0 was published in 1992 and was supported by Microsoft® Windows® 3.1. The final update to APM, 1.2, was published in 1996.

ACPI 1.0 was developed in 1996, and Microsoft first added support for it in Windows NT®. For a period, Windows preferred ACPI, but retained APM support to handle older platforms. With the release of Windows Vista™, Microsoft has completed the transition to ACPI by dropping support for APM.

Many platform vendors deleted their APM support during this transition period, and few will retain APM support now that this transition is complete.

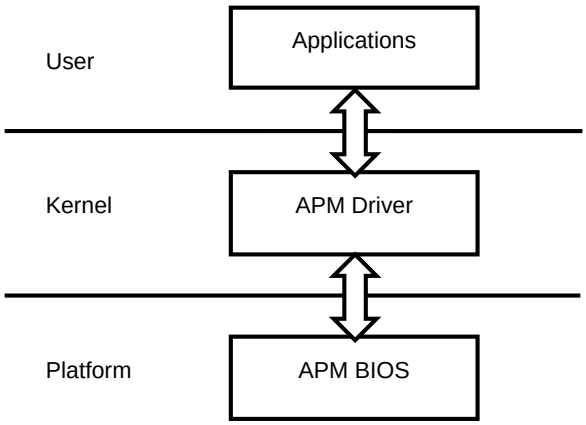


Figure 1: APM Architecture

1.1.1 APM overview

The goal of the APM specification was to extend battery life, while hiding the details of how that is done from the OS in the APM BIOS.

APM defines five general system power states: Full On, APM Enabled, APM Standby, APM Suspend, and Off. The Full On state has no power management. The APM Enabled state may disable some unused devices. The APM Standby state was intended to be a system state with instantaneous wakeup latency. The APM Suspend was optionally suspend-to-RAM, and/or hibernate-to-disk.

APM defines analogous power states for devices: Device On, Device Power Managed, Device Lower Power, and Device Off. Device context is lost in the Device Off state. APM is somewhat vague about whether it is the job of the OS device driver or the APM BIOS to save and restore the device-operational parameters around Device Off.

APM defines CPU Core control states—Full On, Slow Clock, and Off. Interrupts transition the CPU back to Full On instantaneously.

An APM-aware OS has an APM Driver that connects with the APM BIOS. After a connection is established, the APM Driver and APM BIOS "cooperatively perform power management." What this means is that the OS makes calls into the BIOS to discover and modify the default policies of the APM BIOS, and the OS polls the BIOS at least once per second for APM BIOS events.

The APM BIOS can report events to the APM Driver. For example, after detecting an idle period, the APM BIOS may issue a System Standby Request Notification telling the OS that it wants to suspend. The OS must answer by calling a Set Power State function within a certain time. If the OS doesn't answer within the appropriate time, the BIOS may suspend the system anyway. On resume, APM issues a System Standby Resume Notification to let the OS know what happened. This is the OS's opportunity to update its concept of time-of-day.

The OS can disable APM's built-in concept of requesting a suspend or standby, and can instead manually ask APM to perform these transitions on demand.

The OS can instrument its idle loop to call into the APM BIOS to let it know that the processor is idle. The APM BIOS would then perhaps throttle the CPU if it appeared to be running faster than necessary.

The APM BIOS knows about AC/DC status. The APM Driver can query the BIOS for current status, and can also poll for AC/DC change events.

The APM BIOS knows about battery topology and status. The APM Driver can query for configuration as well as capacity, and can poll for Battery Low Notification.

APM supports a hard-coded list of devices for power management including display, disk, parallel ports, serial ports, network adapters, and PCMCIA sockets. The OS can query for their state, enable/disable the APM BIOS from managing the devices, and poll for state changes.

1.1.2 Why APM is not viable

APM is fundamentally centered around the the APM BIOS. The APM BIOS is entered from OS APM Driver calls as well as from System Management Interrupts (SMI) into System Management Mode (SMM). SMM is necessary to implement parts of APM since BIOS code needs to run on the processor without the knowledge or support of the OS.

But it turns out that calling into the BIOS is a really scary thing for an operating system to do. The OS puts the stability of the system in the hands of the BIOS developer on every call. Indeed, the only thing more frightening to the OS is SMM itself, which is completely

transparent to the OS and thus virtually impossible to debug. The largest problem with both of these is that if the state of the system was not as the BIOS developer expected it, then it may not be restored properly on BIOS return to the OS.

So the quality of the "APM experience" varied between platforms depending on the platform's APM BIOS implementation.

Further, the APM specification includes hard-coded limitations about the system device configuration. It is not extensible such that the platform firmware can accommodate system configurations that did not exist when the specification was written.

The philosophy of ACPI, on the other hand, is to put the OS, not the BIOS, in charge of power management policy. ACPI calls this OSPM, or "Operating System-directed configuration and Power Management." OSPM never jumps into the BIOS in ACPI-mode. However, it does access system devices and memory by interpreting BIOS ACPI Machine Language (AML) in kernel mode.

ACPI reduces the the need for SMM, but SMM is still a tool available to BIOS writers to use when they see fit.

ACPI's AML is extensible. It can describe resources and capabilities for devices that the specification knows nothing about—giving the OS the ability to configure and power-manage a broad range of system configurations over time.

ACPI 1.0 was published at the end of 1996. It is probably fair to say that platforms did not universally deploy it until ACPI 2.0 was published in 2000. At that time, Microsoft released Windows 2000, and the APM era was effectively over.

So if you've got a notebook from 1998 or 1999 that includes both APM and ACPI support, you may find that its APM implementation is more mature (and better tested) than its new ACPI implementation. Indeed, it is true that the upstream Linux kernel enables ACPI on all systems that advertise ACPI support, but I recommend that Linux distributors ship with `CONFIG_ACPI_BLACKLIST_YEAR=2000` to disable ACPI in Linux on machines from the last century.

1.2 Multi-Processor Specification (MPS)

MPS 1.1 was issued in 1994. The latest revision, MPS 1.4, was issued in 1995, with minor updates until May, 1997. The primary goal of MPS was to standardize multi-processor configurations such that a "shrink-wrap" OS could boot and run on them without customization. Thus a customer who purchased an MPS-compliant system would have a choice of available Operating Systems.

MPS mandated that the system be symmetric—all processors, memory, and I/O are created equal. It also mandated the presence of Local and I/O APICs.

The Local APIC specified support for inter-processor interrupts—in particular, the INIT IPI and STARTUP IPI used to bring processors on-line.

While the specification calls it an "MP BIOS," the code is much different from the "APM BIOS." The MP BIOS simply puts all the processors into a known state so that the OS can bring them on-line, and constructs static MP configuration data structures—the MP tables—that enumerate the processors and APICS for the OS.

MPS also specified a standard memory map. However, this memory map was later replaced by e820, which is part of the ACPI specification.

The MPS tables enumerate processors, buses, and IO-APICs; and the tables map interrupt sources to IO-APIC input pins.

MPS mandated that SMP siblings be of equal capability. But when Intel introduced Hyper-Threading Technology (HT) with the Pentium® 4 processor, suddenly siblings where not all created equal. What would happen to the installed base if MPS enumerated HT siblings like SMP siblings? Certainly if an HT-ignorant OS treated HT siblings as SMP, it would not schedule tasks optimally.

So the existing MPS 1.4 was not extended to handle HT,[1] and today HT siblings are only available to the OS by parsing the ACPI tables.

But MPS had a sister specification to help machines handle mapping interrupt wires in PIC an IO-APIC mode—the PIRQ spec.

1.3 ACPI vs. PCI IRQ Routers (PIRQ)

IRQ routers are motherboard logic devices that connect physical IRQ wires to different interrupt controller in-

[1] Some BIOSes have a SETUP option to enumerate HT siblings in MPS, but this is a non-standard feature.

put pins. Microsoft published [PIRQ], describing OS-visible tables for PIRQ routers. However, the spec excludes any mention of a standard method to get and set the routers—instead stating that Microsoft will work with the chipset vendors to make sure Windows works with their chipsets.

ACPI generalizes PIRQ routers into ACPI PCI Interrupt Link Devices. In real-life, these are just AML wrappers for both the contents of the PIRQ tables, and the undocumented chipset-specific get/set methods above. The key is that the OS doesn't need any chipset-specific knowledge to figure out what links can be set to, what they are currently set to, or to change the settings. What this means is that the ACPI-aware OS is able to route interrupts on platforms for which it doesn't have intimate knowledge.

1.4 With benefits come risks

It is fair to say that the Linux/ACPI sub-system is large and complex, particularly when compared with BIOS-based implementations such as APM. It is also fair to say that enabling ACPI—effectively an entire suite of features—carries with it a risk of bugs. Indeed it carries with it a risk of regressions, particularly on pre-2000 systems which may have a mature APM implementation and an immature ACPI implementation.

But the hardware industry has effectively abandoned the previous standards that are replaced by ACPI. Further, Linux distributors are now universally shipping and supporting ACPI in Linux. So it is critical that the Linux community continue to build the most robust and full featured ACPI implementation possible to benefit its users.

2 Myth: Suspend to Disk doesn't work, it must be ACPI's fault.

The suspend-to-disk (STD) implementation in Linux has very little to do with ACPI. Indeed, if STD is not working on your system, try it with `acpi=off` or `CONFIG_ACPI=n`. Only if it works better without ACPI can you assume that it is an ACPI-specific issue.

ACPI's role during suspend-to-disk is very much like its role in a normal system boot and a normal system power-off. The main difference is that when ACPI is available, STD uses the "platform" method to power off the machine instead of the "shutdown" method. This allows more devices to be enabled as wakeup devices, as some can wake the system from suspend to disk, but not from soft power-off.

Many end-users think think that STD and ACPI are practically synonyms. One reason for this is because in the past, `/proc/acpi/sleep` was used to invoke STD. However, this method is now deprecated in favor of the generic `/sys/power/state` interface.

Note that suspend-to-RAM (STR) is more dependent on ACPI than STD, as the sleep and wakeup paths are ACPI-specific.

However, the vast majority of both STD and STR failures today have nothing to do with ACPI itself, but instead are related to device drivers. You can often isolate these issues by unloading a device driver before suspend, and re-loading it after resume.

3 Myth: The buttons on my notebook don't work, it must be ACPI's fault.

The ACPI specification standardizes only 3 buttons—power, sleep, and lid. If these buttons do not work in ACPI-mode, then it is, indeed, an ACPI issue.

The other buttons on the keyboard are handled in a variety of platform-specific methods.

First there are the standard keyboard buttons that go directly to the input sub-system. When these malfunction, it cannot be blamed on ACPI, because if there is a problem with them, they'll have the same problem with `acpi=off`.

The "ACPI issues" appear with "hot-keys," which are platform-specific buttons that are not handled by the standard keyboard driver.

When in `acpi=off` mode, these are sometimes handled in the BIOS with SMI/SMM. But when in ACPI-mode, this SMM support is disabled by the platform vendor on the assumption that if ACPI-mode is enabled, then a modern OS sporting a platform-specific hot-key driver is available to handle the hot-keys. For Windows, the vendor may be able to guarantee this is true. However, to date, no platform vendor has volunteered to create or support the community in the creation of platform-specific hot-key drivers for Linux.

Sometimes these keys actually do use the ACPI subsystem to get their work done. However, they report events to vendor-specific ACPI devices, which need vendor-specific ACPI device drivers to receive the events and map them to actions.

4 Myth: My motherboard boots with `acpi=off`, but fails otherwise, it must be ACPI's fault.

With the advent of multi-core processors, SMP systems are very common, and all modern x86 SMP system have an IO-APIC.

However, many notebook and desktop BIOS vendors do not include MPS support in their BIOS. So when booted with `acpi=off`, these systems revert all the way back to 8259 PIC mode. So only in ACPI-mode is the IO-APIC enabled, and thus any IO-APIC mode issues get blamed on ACPI.

The real ACPI vs. non-ACPI, apples-versus-apples comparison would be `acpi=off` vs. `noapic`—for both of these will boot in PIC mode.

But why do so many systems have trouble with IO-APIC mode? The most common reason is the periodic HZ timer. Linux typically uses the 8254 Programmable Interrupt Timer (PIT) for clock ticks. This timer is typically routed to IRQ0 via either IO-APIC pin0 or pin2.

But Windows doesn't always use the PIT; it uses the RTC on IRQ8. So a system vendor that validates their system only with Windows and never even boots Linux before releasing their hardware may not notice that the 8254 PIT used by Linux is not hooked up properly.

5 Myth: The Linux community has no influence on the ACPI Specification.

HP, Intel, Microsoft, Phoenix, and Toshiba co-developed the ACPI specification in the mid-1990s, but it continues to evolve over time. Indeed, version 3.0b was published in October, 2006.

Linux plays a role in that evolution. The latest version of the specification included a number of "clarifications" that were due to direct feedback from the Linux community.

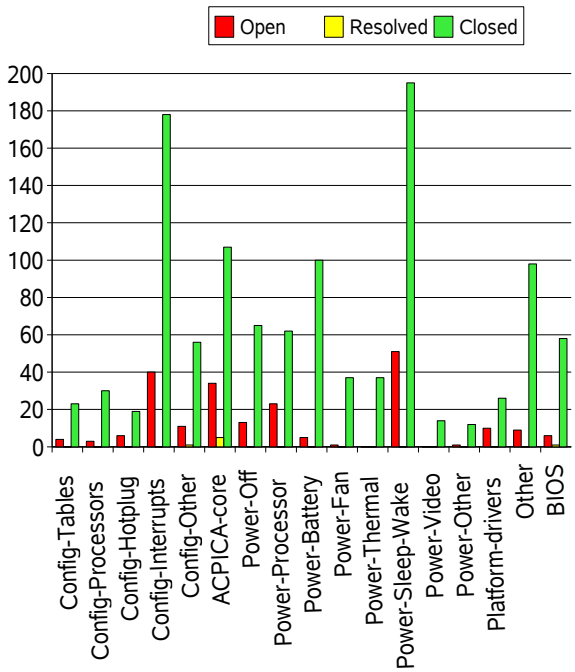


Figure 2: ACPI sighting profile at bugzilla.kernel.org

Sometimes Linux fails to work properly on a system in the field and the root cause is that the ACPI specification was vague. This caused the Linux implementation to do one thing, while the BIOS vendors and Microsoft did another thing.

The Linux/ACPI team in the Intel Open Source Technology Center submits "specification clarifications" directly to the ACPI committee when this happens. The specification is updated, and Linux is changed to match the corrected specification.

6 Myth: ACPI bugs are all due to sub-standard platform BIOS.

When Linux implemented and shipped ACPI, we ran into three categories of failures:

1. Linux fails because the platform BIOS clearly violates the written ACPI specification.

   These failures exist because until Linux implemented ACPI, platform vendors had only Windows OS compatibility tests to verify if their ACPI implementation was correct.

Unfortunately, an OS compatibility test is not a specification compliance test. The result is that many BIOS implementations work by accident because they have been exercised by only one OS.

Today, the Linux-ready Firmware Developer Kit [FWKIT] is available so that vendors who care about Linux have the tools they need to assure their BIOS implementation is compatible with Linux.

2. Linux fails because the platform BIOS writer and Linux programmer interpreted the ACPI specification differently.

   As mentioned in the previous section, we treat these as Linux bugs, fix them, and update the specification to match the actual industry practice.

3. Bugs in the Linux/ACPI implementation. These are simply bugs in Linux, like any other bugs in Linux.

The myth is that a large number of failures are due BIOS bugs in category #1. The reality is shown by Figure 2—under 10% of all Linux/ACPI bugs can be blamed on broken BIOSes.

The majority of bugs have actually been reported against category #3, the Linux-specific code.

7 Myth: ACPI code seems to change a lot, but isn't getting any better.

When ACPI was still new in Linux and few distributors were shipping it, there were many times when changes would fix several machines, but break several others. To be honest, a certain amount of experimentation was going on to figure out how to become bug-compatible with the installed base of systems.

Marcelo Tosatti was maintaining Linux-2.4, and he walked up to me and in a concerned voice asked why we'd break some systems while fixing others. It was clear we needed validation tests to prevent regressions, but we didn't have them yet. And before we did, Linux distributors cut over to Linux-2.6, and almost universally started shipping ACPI. Suddenly we had a large installed base running Linux/ACPI.

For a short while we didn't mend our ways of experimenting on the user base. Then Linus Torvalds scolded us for knowingly causing regressions, insisting that even if a system is working by mistake, changes should never knowingly break the installed base. He was right, of course, and ever since the Linux/ACPI team has made regressions the highest-priority issues.

But while this was happening, a team at Intel was creating three test suites that today are used to to verify that Linux/ACPI is constantly improving.

1. The ACPICA ASL Test Suite (ASLTS) is distributed in open source along with the ACPICA source package on `intel.com`. [ACPICA] ASLTS runs a suite of over 2,000 ASL tests against the ACPICA AML interpreter in a simulation environment. This is the same interpreter that resides in the Linux Kernel. During the development of this test suite, over 300 issues were found. Today there are fewer than 50 unresolved. ACPICA changes are not released if there are any regressions found by this test suite.

2. The ACPICA API Test Suite exercises the interfaces to the ACPICA core as seen from the OS. Like ASLTS, the API tests are done in a simulation environment.

3. ACPI ABAT—Automated Basic Acceptance Tests—which run on top of Linux, exercising user-visible features that are implemented by ACPI. ACPI ABAT is published in open source on the Linux/ACPI home page.[2]

Finally, one can examine the bug profile at `bugzilla.kernel.org` and observe that 80% of all sightings are now closed.

8 Myth: ACPI is slow and thus bad for high-performance cpufreq governors such as "ondemand."

It is true that the BIOS exports AML to the OS, which must use an AML interpreter to parse it. It is true that parsing AML is not intended to occur on performance-critical paths. So how can ACPI possibly be appropriate for enabling P-state transitions such as those made by the high-performance "ondemand" P-state governor—many times per second?

[2] `http://acpi.sourceforge.net`

The answer is that AML is used to parse the ACPI tables to tell cpufreq what states `ondemand` has to choose from. `ondemand` then implements its run-time policy without any involvement from ACPI.

The exception to this rule is that the platform may decide at run-time that the number of P-states should change. This is a relatively rare event, e.g. on an AC→DC transition, or a critical thermal condition. In this case, ACPI re-evaluates the list of P-states and informs cpufreq what new states are available. Cpufreq responds to this change and then proceeds to make its run-time governor decisions without any involvement from ACPI.

9 Myth: Speedstep-centrino is native and thus faster than ACPI-based 'acpi-cpufreq.'

To change the processor frequency and voltage, the OS can either write directly to native, model-specific registers (MSR), or it can access an IO address. There can be a significant efficiency penalty for IO access on some systems, particularly those which trap into SMM on that access.

So the community implemented speedstep-centrino, a cpufreq driver with hard-coded P-state tables based on CPUID and the knowledge of native MSRs. Speedstep-centrino did not need ACPI at all.

At that time, using acpi-cpufreq instead of speedstep-centrino meant using the less-efficient IO accesses. So the myth was true—but two things changed.

1. Speedstep-centrino's hard-coded P-state tables turned out to be difficult to maintain. So ACPI-table capability was added to speedstep-centrino where it would consult ACPI for the tables first, and use the hard-coded tables as a backup.
2. Intel published the "Intel Processor Vendor-Specific ACPI Interface Specification" along with [ACPICA]. This specification made public the bits necessary for an ACPI implementation to use native MSR access. So native MSR access was added to acpi-cpufreq.

The result was that both drivers could talk ACPI, and both could talk to MSRs. Speedstep-centrino still had its hard-coded tables, and acpi-cpufreq could still talk to IO addresses if the system asked it to.

Recently, acpi-cpufreq has been anointed the preferred driver of the two, and speedstep-centrino is scheduled for removal from the source tree as un-maintainable.

10 Myth: More CPU idle power states (C-states) are better than fewer states.

Users observe the system C-states in `/proc/acpi/processor/*/power` and assume that systems with more C-states save more power in idle than systems with fewer C-states. If they look at the data book for an Intel Core™2 Duo processor and try to relate those states to this file, then that is a reasonable assumption.

However, with some limitations, the mapping between hardware C-states and the ACPI C-states seen by Linux is arbitrary. The only things that matter with C-states is the amount of power saved, and the latency associated with waking up the processor. Some systems export lots of C-states, others export fewer C-states and have power-saving features implemented behind the scenes.

An example of this is Dynamic Cache Sizing. This is not under direct OS or C-state control. However, the processor recognizes that when deep C-states are entered, it can progressively flush more and more of the cache. When the system is very idle, the cache is completely flushed and is totally powered off. The user cannot tell if this feature is implemented in the processor by looking at how many C-states are exported to the OS—it is implemented behind the scenes in processor firmware.

11 Myth: Throttling the CPU will always use less energy and extend battery life.

$Energy = Power * Time$. That is to say, *[Watt-Hours] = [Watts] * [Hours].*

Say the processor is throttled to half clock speed so that it runs at half power, but takes twice as long to get the job done. The energy consumed to retire the workload is the same and the only effect was to make the work take twice as long. Energy/work is constant.

There are, however, some second-order effects which make this myth partially true. For one, batteries are not ideal. They tend to supply more total energy when drained at a lower rate than when drained at a higher rate.

Secondly, systems with fans require energy to run those fans. If the system can retire the workload without getting hot, and succeeds in running the fans slower (or off), then less energy is required to retire the workload.

Note that processor clock throttling (ACPI T-states) discussed here should not be confused with processor performance states (ACPI P-states). P-states simultaneously reduce the voltage with the clock speed. As power varies as voltage squared, deeper P-states do take the processor to a more efficient energy/work operating point and minimize energy/work.

12 Myth: I can't contribute to improving ACPI in Linux.

The basis of this last myth may be the existence of ACPICA.

ACPICA (ACPI Component Architecture) is Intel's reference implementation of the ACPI interpreter and surrounding OS-agnostic code. In addition to Linux, BSD®, Solaris™, and other operating systems rely on ACPICA as the core of their ACPI implementation. For this to work, Intel holds the copyright on the code, and publishes under dual BSD and GPL licenses.

In Linux, 160 ACPICA files reside in sub-directories under `/drivers/acpi`. When a patch is submitted from the Linux community to those files, the Linux/ACPI maintainer asks for their permission to license the change to Intel to re-distribute under both licenses on the file, not just the GPL. That way, Intel can share the fix with the other ACPICA users rather than having the multiple copies diverge.

The ACPICA license isn't a barrier for open source contributors, but since it isn't straight GPL and extra permission is requested, it does generate a false impression that patches are unwelcome.

Further, it is the 40 pure-GPL files in `/drivers/acpi` that are most interesting to the majority of the Linux community anyway, for those files contain all the Linux-specific code and ACPI-related policy—treating the ACPICA core as a "black box."

But submitting patches is only one way to help. As described earlier, a lot of the work surrounding Linux/ACPI is determining what it means to be bug-compatible with common industry platform BIOS practice. The more people that are testing and poking at ACPI-related functions on a broad range of systems, the easier it is for the developers to improve the subsystem.

Your system should boot as well (or better) in ACPI-mode using no boot parameters as it does with `acpi=off` or other workarounds. Further, the power management features supported by ACPI such as suspend-to-RAM and processor power management should function properly and should never stop functioning properly.

It is a huge benefit to the community and the quality of the Linux ACPI implementation when users insist that their machines work properly—without the aid of workarounds. When users report regressions, file bugs, and test fixes, they are doing the community a great services that has a dramatic positive impact on the quality of ACPI in Linux.

13 Conclusion

Forget your initial impressions of Linux/ACPI made years ago. ACPI in Linux is not a myth, it is now universally deployed by the major Linux distributors, and it must function properly. Insist that the ACPI-related features on your system work perfectly. If they don't, complain loudly and persistently[3] to help the developers find and fix the issues.

The community must maintain its high standards for ACPI in Linux to continuously improve into the highest quality implementation possible.

References

[ACPI] Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba, *Advanced Configuration and Power Interface*, Revision 3.0b, October, 2006. `http://www.acpi.info`

[ACPICA] Intel, *ACPI Component Architecture*, `http://www.intel.com/technology/iapc/acpi/downloads.htm`

[APM] Intel, Microsoft *Advanced Power Management BIOS Interface Specification*, Revision 1.2, February, 1996. `http://www.microsoft.com/whdc/archive/amp_12.mspx`

[3] Start with `linux-acpi@vger.kernel.org` for Linux/ACPI related issues.

[FWKIT] *Linux-Ready Firmware Developer Kit*, http://www.linuxfirmwarekit.org

[MPS] Intel, *MultiProcessor Specification*, Version 1.4, May, 1997. http://www.intel.com/design/intarch/MANUALS/242016.htm

[PIRQ] Microsoft, *PCI IRQ Routing Table Specification*, Version 1.0, February, 1996. http://www.microsoft.com/whdc/archive/pciirq.mspx

Cool Hand Linux® – Handheld Thermal Extensions

Len Brown
Intel Open Source Technology Center
len.brown@intel.com

Harinarayanan Seshadri
Intel Ultra-Mobile Group
harinarayanan.seshadri@intel.com

Abstract

Linux has traditionally been centered around processor performance and power consumption. Thermal management has been a secondary concern—occasionally used for fan speed control, sometimes for processor throttling, and once in a rare while for an emergency thermal shutdown.

Handheld devices change the rules. Skin temperature is a dominant metric, the processor may be a minority player in heat generation, and there are no fans.

This paper describes extensions to Linux thermal management to meet the challenges of handheld devices.

1 Handhelds Thermal Challenges

The new generation handheld computing devices are exploding with new usage models like navigation, infotainment, health, and UMPC. "Internet on the Go" is a common feature set the handheld devices must provide for all these new usage models. This requires handheld devices to be high-performance.

High-performance handhelds have magnified power and thermal challenges as compared to notebooks.

- Notebooks today are infrequently designed to sit in your lap. Most of them are designed to sit on a table, and many actually require free air flow from beneath the case. Users demand that handheld computers be cool enough that their hand does not sweat. Thus, there are strict skin temperature limits on handhelds.
- Notebooks typically have fans. Handhelds typically do not have fans.
- Handheld form factors are physically smaller than a typical notebook, and thus the thermal dissipation within the platform is limited.
- The CPU is the dominant heat generator on most notebooks. But for handhelds, the CPU may be a minor contributor as compared to other devices.

2 ACPI Thermal Capabilities

Linux notebooks today use a combination of ACPI and native-device thermal control.

The ACPI specification [ACPI] mandates that if a notebook has both active and passive cooling capability, then the platform must expose them to the OS via ACPI. The reasoning behind this is that the OS should be involved in the cooling policy decision when deciding between active versus passive cooling.

But a large number of notebooks implement thermal control "behind the scenes" and don't inform or involve ACPI or the OS at all. Generally this means that they control the fan(s) via chipset or embedded controller; but in some cases, they go further, and also implement thermal throttling in violation of the ACPI spec.

2.1 Linux will not use the ACPI 3.0 Thermal Extensions

ACPI 3.0 added a number of thermal extensions—collectively called the "3.0 Thermal Model." These extensions include the ability to relate the relative contributions of multiple devices to multiple thermal zones. These relative contributions are measured at system design-time, and encoded in an ACPI thermal relationship table for use by the OS in balancing the multiple contributors to thermal load. This is a sophisticated solution requiring thorough measurements by the system designer, as well as knowledge on the part of the OS about relative performance tradeoffs.

The handheld effort described in this paper does not need the ACPI 3.0 thermal extensions. Indeed, no Linux

notebook has yet materialized that needs those extensions.

So when the BIOS AML uses `_OSI` and asks Linux if it supports the "3.0 Thermal Model," Linux will continue to answer "no," for the immediate future.

2.2 How the ACPI 2.0 Thermal Model works

ACPI defines a concept of thermal zones. A thermal zone is effectively a thermometer with associated trip points and devices.

A single CRT trip point provokes a critical system shutdown. A single HOT trip point provokes a system suspend-to-disk. A single PSV trip point activates the OS's passive cooling algorithm. One or more ACx trip points control fans for active cooling. Each active trip point may be associated with one or multiple fans, or with multiple speeds of the same fan.

When a PSV trip point fires, the Linux `processor_thermal` driver receives the event and immediately requests the Linux cpufreq subsystem to enter the deepest available processor performance state (P-state). As P-states reduce voltage along with frequency, they are more power-efficient than simple clock throttling (T-states), which lower frequency only.

The processor thermal throttling algorithm then periodically polls the thermal zone for temperature, and throttles the clock accordingly. When the system has cooled and the algorithm has run its course, the processor is un-throttled, and cpufreq is again allowed to control P-states.

2.3 Why is the ACPI 2.0 Thermal Model not sufficient?

It can't hurt to implement ACPI's CRT trip point for critical system shutdown—but that isn't really the focus here.

The HOT trip point doesn't really make sense, since on a handheld, shutdown and hibernate to disk (if one even exists) are likely to be synonymous.

Active trip points are of no use on systems which have no fans.

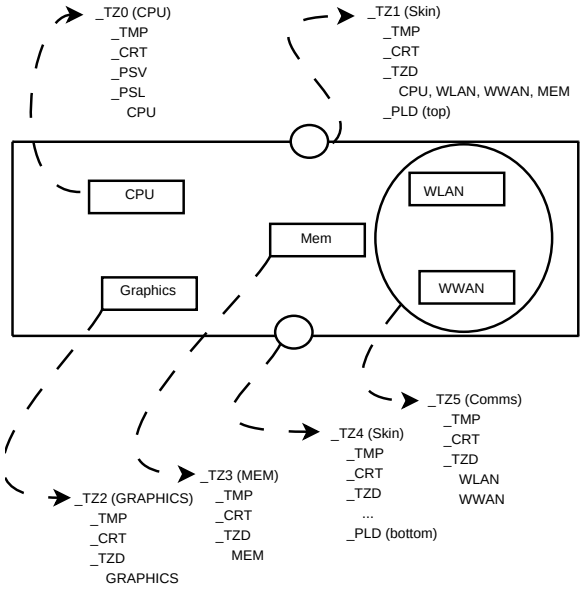


Figure 1: Example Sensor-to-Thermal-Zone Mapping

That leaves the single PSV trip point. ACPI 2.0 can associate (only) a processor throttling device with a trip point. Yes, handhelds have processors, but the processor isn't expected to always be the dominant contributor to thermal footprint on handhelds like it often is on notebooks.

ACPI 2.0 includes the `_TZD` method to associate devices with thermal zones. However, ACPI doesn't say anything about how to throttle non-processor devices—so that must be handled by native device drivers.

2.4 So why use ACPI at all?

Inexpensive thermal sensors do not know how to generate events. They are effectively just thermometers that need to be polled. However, using the CPU to poll the sensors would be keeping a relatively power-hungry component busy on a relatively trivial task. The solution is to poll the sensors from a low-power embedded controller (EC).

The EC is always running. It polls all the sensors and is responsible for interrupting the main processor with events. ACPI defines a standard EC, and so it is easy to re-use that implementation. Of course, it would be an equally valid solution to use a native device driver to talk to an on-board microcontroller that handles the low-level polling.

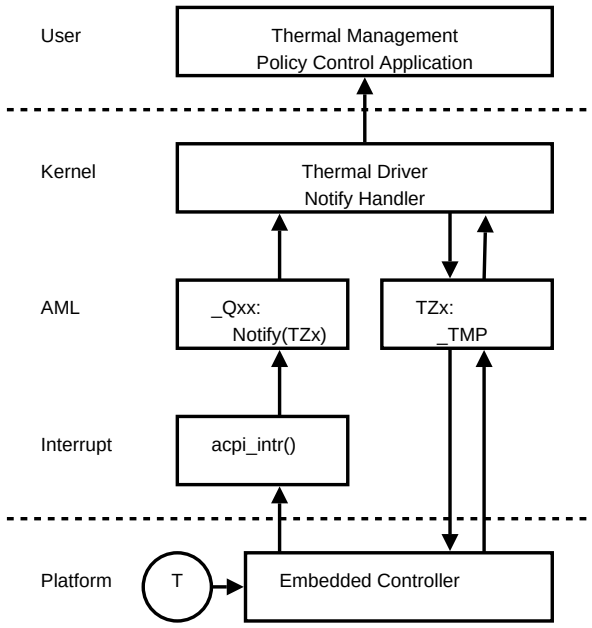


Figure 2: Thermal Event Delivery via ACPI

2.5 Mapping Platform Sensors to ACPI Thermal Zones

Figure 1 shows an example of mapping platform sensors to the ACPI Thermal zones. Four different scenarios are illustrated in the figure:

- Sensors that are built into the CPU.
- Sensors that are associated with a single non-processor device, such as DRAM or graphics.
- Sensors that are associated with cluster of components, for example, Wireless LAN (WLAN) and Wireless WAN (WWAN).
- Skin sensors that indicate overall platform temperature.

2.6 How to use ACPI for Handheld Thermal Events

For the CPU, Linux can continue to handle a single ACPI passive trip point with an in-kernel processor thermal throttling algorithm.

For critical thermal events, Linux can continue to handle a single ACPI critical trip point with a system shutdown.

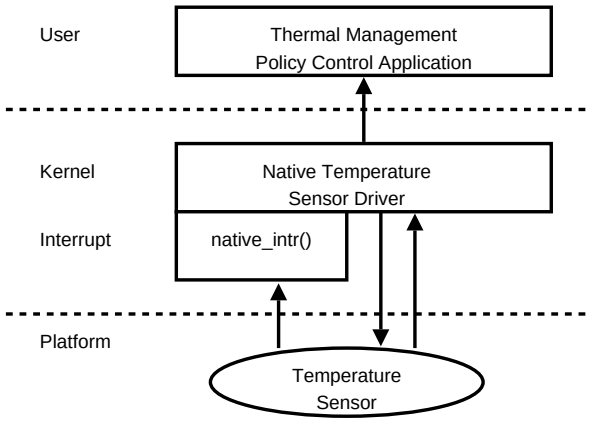


Figure 3: Thermal Event Delivery via native driver

However, for the non-processor thermal zones, a single passive trip point is insufficient. For those we will use ACPI's concept of "temperature change events."

When the EC decides that the temperature has changed by a meaningful amount—either up or down—it sends a temperature change event to the thermal zone object.

If the thermal zone is associated with a processor, then the kernel can invoke its traditional processor thermal throttling algorithm.

As shown in Figure 2, for non-processor thermal zones, the thermal driver will query the temperature of the zone, and send a netlink message to user-space identifying the zone and the current temperature.

Figure 3 shows the same event delivered by a native platform, sensor-specific sensor driver.

3 Proposed Handheld Thermal Solution

Multiple works ([LORCH], [VAHDAT]) focus on low-power OS requirements and low-power platform design. While low-power optimizations have a positive impact on thermals, this only addresses thermal issues at a component level. To address the platform-level thermal issues, we need to look at the thermal problem in a more complete platform perspective. This requires support from OS, user applications, etc.

3.1 Design Philosophy

- Thermal monitoring will be done using inexpensive thermal sensors—polled by a low-power EC.

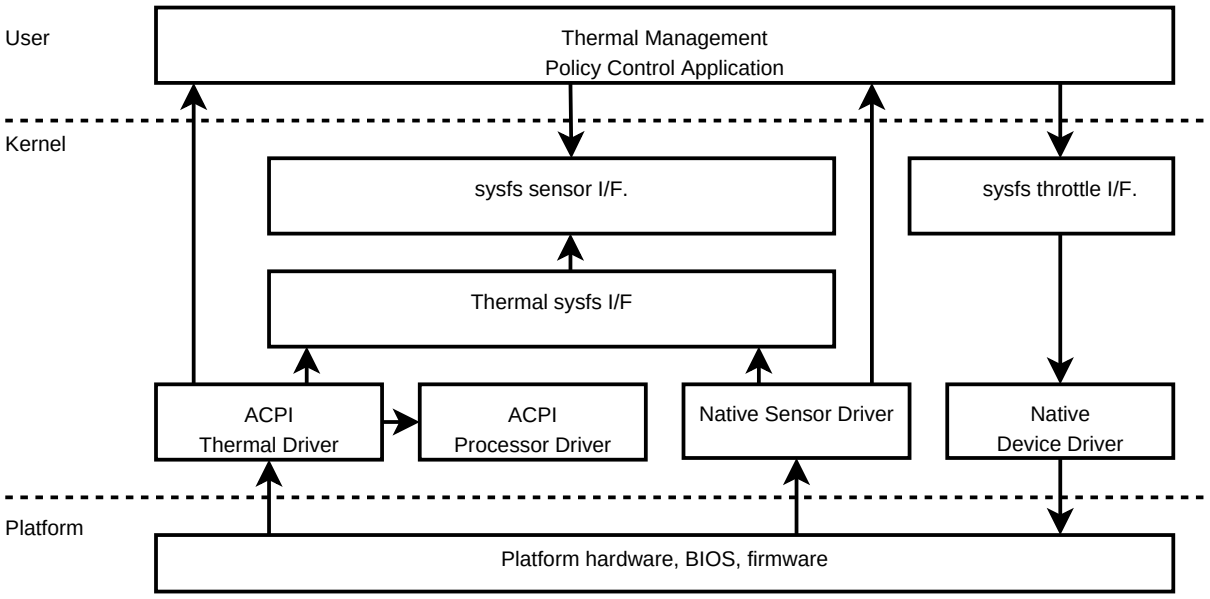


Figure 4: Thermal Zone Driver Stack Architecture

- Thermal management policy decisions will be made from user space, as the user has a comprehensive view of the platform.

- The kernel provides only the mechanism to deliver thermal events to user space, and the mechanism for user space to communicate its throttling decisions to native device drivers.

Figure 4 shows the thermal control software stack. The thermal management policy control application sits on top. It receives netlink messages from the kernel thermal zone driver. It then implements device-specific thermal throttling via sysfs. Native device drivers supply the throttling controls in sysfs and implement device-specific throttling functions.

3.2 Thermal Zone module

The thermal zone module has two components—a thermal zone sysfs driver and thermal zone sensor driver. The thermal zone sysfs driver is platform-independent, and handles all the sysfs interaction. The thermal zone sensor driver is platform-dependent. It works closely with the platform BIOS and sensor driver, and has knowledge of sensor information in the platform.

3.2.1 Thermal sysfs driver

The thermal sysfs driver exports two interfaces (`thermal_control_register()` and `thermal_control_deregister()`) to component drivers, which the component drivers can call to register their control capability to the thermal zone sysfs driver.

The thermal sysfs driver also exports two interfaces—`thermal_sensor_register()` and `thermal_sensor_deregister()`—to the platform-specific sensor drivers, where the sensor drivers can use this interface to register their sensor capability.

This driver is responsible for all thermal sysfs entries. It interacts with all the platform specific thermal sensor drivers and component drivers to populate the sysfs entries.

The thermal zone driver also provides a notification-of-temperature service to a component driver. The thermal zone sensor driver as part of registration exposes its sensing and thermal zone capability.

3.2.2 Thermal Zone sensor driver

The thermal zone sensor driver provides all the platform-specific sensor information to the thermal

sysfs driver. It is platform-specific in that it has prior information about the sensors present in the platform.

The thermal zone driver directly maps the ACPI 2.0 thermal zone definition, as shown in Figure 1. The thermal zone sensor driver also handles the interrupt notification from the sensor trips and delivers it to user space through netlink socket.

3.3 Component Throttle driver

All the component drivers participating in the given thermal zone can register with the thermal driver, each providing the set of thermal ops it can support. The thermal driver will redirect all the control requests to the appropriate component drivers when the user programs the throttling level. Its is up to the component driver to implement the thermal control.

For example, a component driver associated with DRAM would slow down the DRAM clock on throttling requests.

3.4 Thermal Zone Sysfs Property

Table 1 shows the directory structure exposing each thermal zone sysfs property to user space.

The intent is that any combination of ACPI and native thermal zones may exist on a platform, but the generic sysfs interface looks the same for all of them. Thus, the syntax of the files borrows heavily from the Linux `hwmon` sub-system.[1]

Each thermal zone provides its current temperature and an indicator that can be used by user-space to see if the current temperature has changed since the last read.

If a critical trip point is present, its value is indicated here, as well as an alarm indicator showing whether it has fired.

If a passive trip point is present, its value is indicated here, as well as an alarm indicator showing whether it has fired.

There are symbolic links to the device nodes of the devices associated with the thermal zone. Those devices will export their throttling controls under their device nodes.

[1] `Documentation/hwmon/sysfs-interface` defines the names of the sysfs files, except for the passive files, which are new.

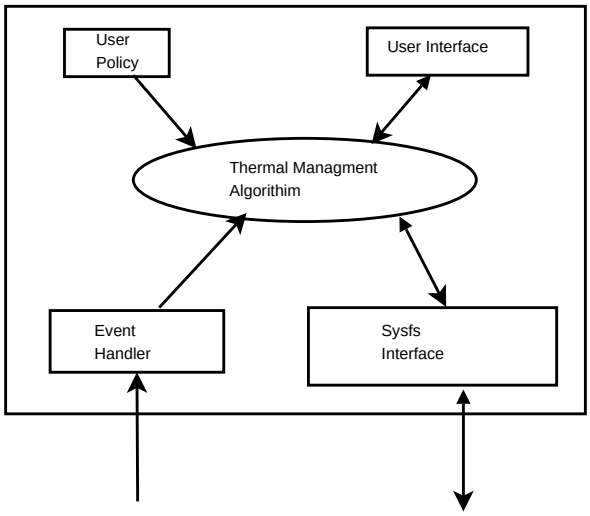


Figure 5: Thermal Policy Control Application

3.5 Throttling Sysfs Properties

Devices that support throttling will have two additional properties associated with the device nodes: `throttling` and `throttling_max`.

A value of 0 means maximum performance, though no throttling. A value of `throttling_max` means maximum power savings in the deepest throttling state available before device state is lost.

3.6 Netlink Socket Kobject event notification

Events will be passed from the kernel to user-space using the Linux netlink facility. Interrupts from the sensor or EC are delivered to user-space through a netlink socket.

3.7 Policy Control Application

Figure 5 shows a thermal policy control application.

The control application interacts with the user, via GUI or configuration files, etc., such that it can understand both the dependencies within the system, and the desired current operating point of the system.

The thermal management module can monitor the platform and component temperature through the sysfs interface, which is a simple wrapper around the sysfs layer. The application receives temperature change

sysfs	ACPI	Description	R/W
temp1_input	_TMP	Current temperature	RO
temp1_alarm		Temperature change occurred	RW
temp1_crit	_CRT	Critical alarm temperature	RO
temp1_crit_alarm		Critical alarm occurred	RW
temp1_passive	_PSV	Passive alarm temperature	RO
temp1_passive_alarm		Passive alarm occurred	RW
<device_name1>		Link to device1 associated with zone	RO
<device_name2>		Link to device2 associated with zone	RO
...		...	RO

Table 1: Thermal Zone sysfs entries

events via netlink, so it can track temperature trends. When it decides to implement throttling, it accesses the appropriate native device's sysfs entries via the sysfs interface.

The thermal throttling algorithms implemented internally to the policy control application are beyond the scope of this paper.

3.8 Possible ACPI extensions

This proposal does make use of the ACPI critical trip point. Depending on the device, the policy manager may decide that either the device or the entire system must be shut down in response to a critical trip point.

This proposal also retains the ACPI 2.0 support for a passive trip point associated with a processor, and in-kernel thermal throttling of the processor device.

However, the main use of ACPI in this proposal is simply as a conduit that associates interesting temperature change events with thermal zones.

What is missing from ACPI is a way for the policy manager to tell the firmware via ACPI what events are interesting.

As a result, the EC must have built-in knowledge about what temperature change events are interesting across the operating range of the device.

However, it would be more flexible if the policy control application could simply dictate what granularity of temperature change events it would like to see surrounding the current temperature.

For example, when the temperature is 20C, the policy application may not care about temperature change events smaller than 5C. But when the temperature is higher, change events of 0.5C may be needed for fine control of the throttling algorithm.

4 Conclusion

On handheld computers it is viable for a platform-specific control application to manage the thermal policy. With the policy moved to user-space, the kernel component of the solution is limited to delivering events and exposing device-specific throttling controls.

This approach should be viable on a broad range of systems, both with and without ACPI support.

References

[ACPI] Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba, *Advanced Configuration and Power Interface 3.0b*, October, 2006 `http://www.acpi.info`

[LORCH] J. Lorch and A.J. Smith. *Software Strategies for Portable Computer Energy Management*, IEEE Personal Communications Magazine, 5(3):60-73, June 1998.

[VAHDAT] Vahdat, A., Lebeck, A., and Ellis, C.S. 2000. *Every joule is precious: the case for revisiting operating system design for energy efficiency*, Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges For the Operating System, Kolding, Denmark, September 17–20, 2000. EW 9. ACM Press, New York, NY, 31–36. `http://doi.acm.org/10.1145/566726.566735`

Asynchronous System Calls

Genesis and Status

Zach Brown
Oracle
zach.brown@oracle.com

Abstract

The Linux kernel provides a system call interface for asynchronous I/O (AIO) which has not been widely adopted. It supports few operations and provides asynchronous submission and completion of those operations in limited circumstances. It has succeeded in providing cache-avoiding reads and writes to regular files, used almost exclusively by database software. Maintaining this minimal functionality has proven to be disproportionately difficult which has in turn discouraged adding support for other operations.

Recently Ingo Molnar introduced a subsystem called *syslets* [3]. Syslets give user space a new system call interface for performing asynchronous operations. They support efficient asynchronous submission and completion of almost every existing system call interface in the kernel.

This paper documents the path that leads from the limits of the existing AIO implementation to syslets. Syslets have the potential to both reduce the cost and broaden the functionality of AIO support in the Linux kernel.

1 Background

Before analyzing the benefits of syslets we first review the motivation for AIO and explain the limitations of the kernel's current support for AIO.

1.1 Briefly, Why AIO?

AIO can lead to better system resource utilization by letting a single process do independent work in parallel.

Take the trivial example of calculating the cryptographic hash of a very large file. The file is read in pieces into memory and each piece is hashed. With synchronous `read()` calls, the CPU is idle while each piece is read from the file. Then as the CPU hashes the file in memory the disk is idle. If it takes the same amount of time to read a piece as it takes to calculate its hash (a bit of a stretch these days) then the system is working at half capacity.

If AIO is used, our example process fully utilizes both the CPU and the disk by letting it issue the read for the next piece without blocking the CPU. After issuing the read, the process is free to use the CPU to hash the current piece. Once this hashing is complete the process finds that the next read has completed and is available for hashing.

The general principles of this trivial example apply to more relevant modern software systems. Trade a streaming read from a large file for random reads from a block device and trade hashing for non-blocking network IO and you have an ISCSI target server.

1.2 KAIO Implementation Overview

The kernel provides AIO for file IO with a set of system calls. These system calls and their implementation in the kernel have come to be known as KAIO. Applications use KAIO by first packing the arguments for IO operations into `iocb` structs. IO operations are initiated by passing an array of `iocb`s, one element per IO operation, to `io_submit()`. Eventually the result of the operations are made available as an array of `io_event` structures—one for each completed IO operation.

In the kernel, the `io_submit()` system call handler translates each of the `iocb`s from user space into a `kiocb` structure—a kernel representation of the pending operation. To initiate the IO, the synchronous file IO code paths are called. The file IO paths have two choices at this point.

The first option is for a code path to block processing the IO and only return once the IO is complete. The process calling `io_submit()` blocks and when it eventually returns, the IO is immediately available for `io_getevents()`. This is what happens if a buffered file IO operation is initiated with KAIO.

The second option is for the file IO path to note that it is being called asynchronously and take action to avoid blocking the caller. The KAIO subsystem provides some infrastructure to support this. The file IO path can return a specific error, `-EIOCBQUEUED`, to indicate that the operation was initiated and will complete in the future. The file IO path promises to call `aio_complete()` on the `kiocb` when the IO is complete.

The O_DIRECT file IO path is the most significant mainline kernel path to implement the second option. It uses block IO completion handlers to call `aio_complete()` after returning `-EIOCBQUEUED` rather than waiting for the block IO to complete before returning.

1.3 KAIO Maintenance Burden

The KAIO infrastructure has gotten us pretty far but its implementation imposes a significant maintenance burden on code paths which support it.

- Continuation can't reference the submitting process. Kernel code has a fundamental construct for referencing the currently executing process. KAIO requires very careful attention to when this is done. Once the handler returns `-EIOCBQUEUED` then the submission system call can return and kernel code will no longer be executing in the context of the submitting process. This means that an operation can only avoid blocking once it has gotten everything it needs from the submitting process. This keeps O_DIRECT, for example, from avoiding blocking until it has pinned all of the pages from user space. It performs file system block offset lookups in the mean time which it must block on.

- Returning an error instead of blocking implies far-reaching changes to core subsystems. A large number of fundamental kernel interfaces block and currently can't return an error. `lock_page()` and `mutex_lock()` are only two of the most important. These interfaces have to be taught to return an error instead of blocking. Not only does this push changes out to core subsystems, it requires rewriting code paths to handle errors from these functions which might not have handled errors before.

- KAIO's special return codes must be returned from their source, which has promised to call `aio_complete()`, all the way back to `io_setup()`, which will call `aio_complete()` if it **does not** see the special error codes. Code that innocently used to overwrite an existing error code with its own, say returning `-EIO` when O_SYNC metadata writing fails, can lead to duplicate calls to `aio_complete()` . This invariant must be enforced through any mid-level helpers that might have no idea that they're being called in the path between `io_submit()` and the source of KAIO's special error codes.

- `iocb`s duplicate system call arguments. For any operation to be supported by KAIO it must have its arguments expressed in the `iocb` structure. This duplicates all the problems that the system call interface already solves. Should we use native types and compatibility translation between user space and kernel for different word sizes? Should we use fixed-width types and create subtle inconsistencies with the synchronous interfaces? If we could reuse the existing convention of passing arguments to the kernel, the system call ABI, we'd avoid this mess.

Far better would be a way to provide an asynchronous system call interface without having to modify, and risk breaking, existing system call handlers.

2 Fibrils

2.1 Blame Linus

In November of 2005, Linus Torvalds expressed frustration with the current KAIO implementation. He suggested an alternative he called *micro-threads*. It built on two fundamental, and seemingly obvious, observations:

1. The C stack already expresses the partial progress of an operation much more naturally than explicit progress storage in `kiocb` ever will.

2. `schedule()`, the core of the kernel task scheduler, already knows when an operation blocks for any reason. Handling blocked operations in the scheduler removes the need to handle blocking at every potential blocking call site.

The proposal was to use the call stack as the representation of a partially completed operation. As an operation is submitted its handler would be executed as normal, exactly as if it was executed synchronously. If it blocked, its stack would be saved away and the stack of the next runnable operation would be put into place.

The obvious way to use a scheduler to switch per-operation stacks is to use a full kernel thread for each operation. Originally it was feared that a full kernel thread per operation would be too expensive to manage and switch between because the existing scheduler interface would have required initiating each operation in its own thread from the start. This is wasted effort if it turns out that the operations do not need their own context because they do not block. Scheduling stacks only when an operation blocks defers the scheduler's work until it is explicitly needed.

After experience with KAIO's limitations, scheduling stacks offers tantalizing benefits:

- The submitting process never blocks
- Very little cost is required to issue non-blocking operations through the AIO interface

Most importantly, it requires no changes to system call handlers—they are almost all instantly supported.

2.2 Fibrils prototype

There were two primary obstacles to implementing this proposal.

First, the kernel associates a structure with a given task, called the `task_struct`. By convention, it's considered private to code that is executing in the context of that task. The notion of scheduling stacks changes this fundamental convention in the kernel. Scheduling stacks, even if they're not concurrently executing on multiple CPUs, implies that code which accesses `task_struct` must now be re-entrant. Involuntary preemption greatly increases the problem. Every member of `task_struct` would need to be audited to ensure that concurrent access would be safe.

Second, the kernel interfaces for putting a code path to sleep and waking it back up are also implemented in terms of these `task_structs`. If we now have multiple code paths being scheduled as stacks in the context of one task then we have to rework these interfaces to wake up the stacks instead of the `task_struct` that they all belong to. These sleeping interfaces are some of the most used in the kernel. Even if the changes are reasonably low-risk this is an incredible amount of code churn.

It took about a year to get around to seriously considering making these large changes. The result was a prototype that introduced a saved stack which could be scheduled under a task, called a *fibril* [1].

3 Syslets

The fibrils prototype succeeded in sparking debate of generic AIO system calls. In his response to fibrils [4], Ingo Molnar reasonably expressed dissatisfaction with the fibrils construct. First, the notion of a secondary simpler fibrils scheduler will not last over time as people ask it to take on more functionality. Eventually it will end up as complicated as the task scheduler it was trying to avoid. There were already signs of this in the lack of support for POSIX AIO's prioritized operations. Second, effort would be better spent adapting the main task scheduler to support asynchronous system calls instead of creating a secondary construct to avoid the perceived cost of the task scheduler.

Two weeks later, he announced [2] an interface and scheduler changes to support asynchronous system calls which he called *syslets*.[1]

The syslet implementation first makes the assertion that the modern task scheduler is efficient enough to be used for swapping blocked operations in and out. It uses full kernel tasks to express each blocked operation.

The syslet infrastructure modifies the scheduler so that each operation does not require execution in a dedicated task at the time of submission. If a task submits a system call with syslets and the operation blocks, then the

[1] The reader may be forgiven for giggling at the similarity to Chicklets, a brand of chewing gum.

scheduler performs an implicit `clone()`. The submission call then returns as the **child** of the submitting task. This is carefully managed so that the user space registers associated with the submitting task are migrated to the new child task that will be returning to user space. This requires a very small amount of support code in each architecture that wishes to support syslets.

Returning from a blocked syslet operation in a cloned child is critical to the simplicity of the syslets approach. Fibrils tried to return to user space in the same context that submitted the operation. This lead to extensive modifications to allow a context to be referenced by more than one operation at a time. The syslet infrastructure avoids these modifications by declaring that a blocked syslet operation will return to user space in a new task.

This is a viable approach because nearly all significant user space thread state is either shared between tasks or is inherited by a new child task from its parent. It will be very rare that user space will suffer ill effects of returning in a new child task. One consequence is that `gettid()` will return a new value after a syslet operation blocks. This could require some applications to more carefully manage per-thread state.

4 Implementing KAIO with syslets

So far we've considered asynchronous system calls, both fibrils and syslets, which are accessible through a set of new system calls. This gives user space a powerful new tool, but it does nothing to address the kernel maintenance problem of the KAIO interface. The KAIO interface can be provided by the kernel but implemented in terms of syslets instead of `kiocb`s.

As the `iocb` structures are copied from user space their arguments would be used to issue syslet operations. As the system call handler returns in the syslet thread it would take the result of the operation and insert it into the KAIO event completion ring.

Since the syslet interface performs an implicit clone we cannot call the syslet submission paths directly from the submitting user context. Current KAIO users are not prepared to have their thread context change under them. This requires worker threads which are very carefully managed so as not to unacceptably degrade performance.

Cancellation would need to be supported. Signals could be sent to the tasks which are executing syslets which could cause the handler to return. The same accounting which associated a user's `iocb` with a syslet could be annotated to indicate that the operation should complete as canceled instead of returning the result of the system call.

4.1 Benefits

Implementing KAIO with syslets offers to simplify existing paths which support KAIO. Those paths will also provide greater KAIO support by blocking less frequently.

System call handlers would no longer need to know about `kiocb`s. They could be removed from the file IO paths entirely. Synchronous file IO would no longer need to work with these `kiocbs` when they are not providing KAIO functionality. The specific error codes that needed to be carefully maintained could be discarded.

KAIO submission would not block as often as it does now. As explored in our trivial file hashing example, blocking in submission can lead to resource underutilization. Others have complained that it can make it difficult for an application to measure the latency of operations which are submitted concurrently. This has been observed in the field as O_DIRECT writes issued with KAIO block waiting for an available IO request structure.

4.2 Risks

Implementing KAIO with syslets runs the risk of adding significant memory and CPU cost to each operation. It must be very carefully managed to keep these costs under control.

Memory consumption will go up as each blocked operation is tracked with a kernel thread instead of a `kiocb`. This may be alleviated by limiting the number of KAIO operations which are issued as syslets at any given time. This measure would only be possible if KAIO continues to only support operations which are guaranteed to make forward progress.

Completion will require a path through the scheduler. O_DIRECT file IO demonstrates this problem. Previously it could call `aio_complete()` from block IO

completion handlers which would immediately queue the operation for collection by user space. With syslets the block IO completion handlers would wake the blocked syslet executing the IO. The syslet would wake up and run to completion and return at which point the operation would be queued for collection by user space.

4.3 Remaining Work

An initial rewrite of the KAIO subsystem has been done and put through light testing. At the time of this writing conclusive results are not yet available. There is much work yet to be done. Results may be found in the future on the web at `http://oss.oracle.com/~zab/kaio-syslets/`.

5 Conclusion

KAIO has long frustrated the kernel community with its limited functionality and high maintenance cost. Syslets offer a powerful interface for user space to move towards in the future. With luck, syslets may also ease the burden of supporting existing KAIO functionality while addressing significant limitations of the current implementation.

6 Acknowledgments

Thanks to Valerie Henson and Mark Fasheh for their valuable feedback on this paper.

References

[1] Zach Brown. Generic aio by scheduling stacks. `http://lkml.org/lkml/2007/1/30/330`.

[2] Ingo Molnar. Announce: Syslets, generic asynchronous system call support. `http://lkml.org/lkml/2007/2/13/142`.

[3] Ingo Molnar. downloadable syslets patches. `http://people.redhat.com/mingo/syslet-patches/`.

[4] Ingo Molnar. in response to generic aio by scheduling stacks. `http://lkml.org/lkml/2007/1/31/34`.

Frysk 1, Kernel 0?

Andrew Cagney
Red Hat Canada, Inc.
cagney@redhat.com

Abstract

Frysk is a user-level, always-on, execution analysis and debugging tool designed to work on large applications running on current Linux kernels. Since Frysk, internally, makes aggressive use of the `utrace`, `ptrace`, and `/proc` interfaces, Frysk is often the first tool to identify regressions and problems in those areas. Consequently, Frysk, in addition to its GNOME application and command line utilities, includes a kernel regression test-suite as part of its installation.

This paper will examine Frysk's approach to testing, in particular the development and inclusion of unit tests directly targeted at kernel regressions. Examples will include a number of recently uncovered kernel bugs.

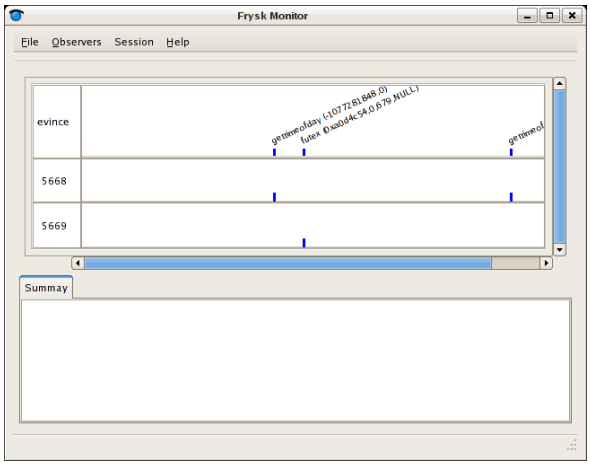


Figure 1: Frysk's Monitor Tool

1 Overview

This paper will first present an overview of the Frysk project, its goals, and the technical complexities or risks of such an effort.

The second section will examine in more detail one source of technical problems or risk—Linux's `ptrace` interface—and the way that the Frysk project has addressed that challenge.

In the concluding section, several specific examples of problems (both bugs and limitations) in the kernel that identified will be discussed.

2 The Frysk Project

The principal goal of the Frysk Project is to develop a suite of always-on, or very-long-running, tools that allow the developer and administrator to both monitor and debug complex modern user-land applications. Further, by exploiting the flexibility of the underlying Frysk architecture, Frysk also makes available a collection of more traditional debugging utilities.

In addition, as a secondary goal, the Frysk project is endeavoring to encourage the advancement of debug technologies, such as kernel-level monitoring and debugging interface (e.g., `utrace`), and debug libraries (e.g., `libdwfl` for DWARF debug info), available to Linux users and developers.

2.1 Frysk's Tool Set

A typical desktop or visual user of Frysk will use the monitoring tool to watch their system, perhaps focusing on a specific application (see Figure 1).

When a problem is noticed Frysk's more traditional debugging tool can be used to drill down to the root-cause (see Figure 2).

For more traditional command-line users, there are Frysk's utilities, which include:

- `fstack` – display a stack back-trace of either a running process or a core file;
- `fcore` – create a core file from a running program;

```
$ fcatch /usr/lib/frysk/funit-stackframe
fcatch: from PID 17430 TID 17430:
SIGSEGV detected - dumping stack trace for TID 17430
#0 0x0804835c in global_st_size () from: \ldots/funit-stackframe.S#50
#1 0x08048397 in main () from: \ldots/funit-stackframe.S#87
#2 0x00c2d4e4 in __libc_start_main ()
#3 0x080482d1 in _start ()
```

Figure 3: Running the `fcatch` utility

```
$ fstep -s 1 ls
[3299]  0xbfb840        mov    %esp,%eax
[3299]  0xbfb842        call   0xbfbf76
[3299]  0xbfbf76        push   %ebp
[3299]  0xbfbf77        mov    %esp,%ebp
[3299]  0xbfbf79        push   %edi
[3299]  0xbfbf7a        push   %esi
....
```

Figure 4: Running the `fstep` utility

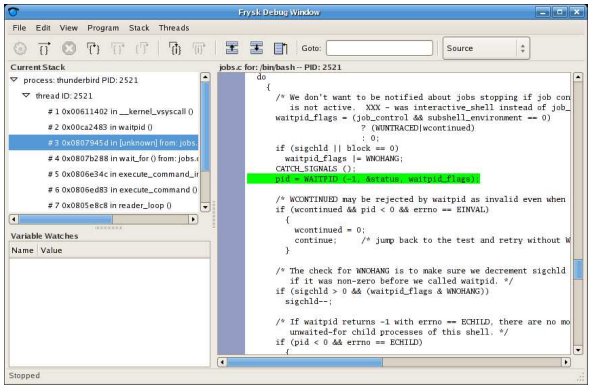


Figure 2: Frysk's Debugger Tool

- `fstep` – instruction-step trace a running program (see Figure 4);
- `fcatch` – catch a program as it is crashing, and display a stack back-trace of the errant thread (see Figure 3);

and of course the very traditional:

- `fhpd` – command-line debugger.

3 The Frysk Architecture

Internally, Frysk's architecture consists of three key components:

- kernel interface – handles the reception of system events (e.g., clone, fork, exec) reported to Frysk by the kernel; currently implemented using `ptrace`.
- the core – the engine that uses the events supplied by the kernel to implement an abstract model of the system.
- utilities and GNOME interface – implemented using the core.

4 Project Risks

From the outset, a number of risks were identified with the Frysk project. Beyond the technical complexities of building a monitoring and debugging tool-set, the Frysk project has additional "upstream" dependencies that could potentially impact on the project's effort:

- `gcj` – the Free GNU Java compiler; Frysk chose to use `gcj` as its principal compiler.
- `Java-GNOME` bindings – used to implement a true GNOME interface; Frysk is recognized as an early adopter of the Java-GNOME bindings.
- Kernel's `ptrace` and `/proc` interfaces – currently used by Frysk to obtain system information;

it was recognized that Frysk would be far more aggressive in its use of these interfaces than any existing clients and hence was likely to uncover new problems and limitations.

- Kernel's `utrace` interface – the maturing framework being developed to both replace and extend `ptrace`

The next two sections will review the Frysk–Kernel interface, problems that were encountered, and the actions taken to address those problems.

5 Managing Project Stability—Testing

One key aspect of a successful project is its overall stability, to that end the quality of testing is a key factor. This section will identify development processes that can both help and hinder that goal, with specific reference to the kernel and its `ptrace` and `/proc` interfaces.

5.1 Rapid Feedback—Linux Kernel

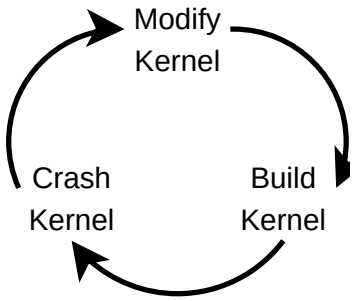


Figure 5: Short Feedback Cycle—Kernel

Linux's rapid progress is very much attributed to its open-source development model. In addition, and as illustrated by Figure 5, the Linux developer and the project's overall stability also benefits from very short testing and release cycles—just the act of booting the modified kernel is a significant testing step. Only occasionally do problems escape detection in the initial testing and review phases.

5.2 Slow Feedback—`ptrace` Component Clients

In contrast, as illustrated by Figure 6, the short development and testing cycle fails when a component is only occasionally exposed to use by its clients. The long lead-in time and the great number of changes that occur before an under-used kernel component is significantly tested greatly increases the risk that the kernel component will contain latent problems.

The `ptrace` interface provides a good illustration of this problem. Both changes directly to that module, such as a new implementation like `utrace`, or indirectly, such as modifications to the `exec` code, can lead to latent problems or limitations that are only detected much later when the significantly modified kernel is deployed by a distribution. Specific examples of this are discussed further in the third section.

5.3 Quickening the Feedback

Frysk, being heavily dependent on both the existing `ptrace` interface and the new `utrace` interface, recognized its exposure very early on in its development. To mitigate the risk of that exposure, Frysk implemented automated testing at three levels:

1. Integration test (Frysk) – using Dogtail (a test framework for GUI applications) and ExpUnit (an `expect` framework integrated into `JUnit`), test the user-visible functionality of Frysk.

2. Unit test (Frysk) – applying test-driven development ensured the rapid creation of automated tests that exercised Frysk's internal interfaces and external functionality; the unit tests allow Frysk developers to quickly isolate a new problem down to a sub-system and then, possibly, that sub-system's interaction with the kernel.

3. Regression test ("upstream") – where the root cause of a problem was determined to be an "upstream" or system component on which Frysk depended (e.g., kernel, compiler, library), a standalone automated test demonstrating the specific upstream problem was implemented.

Further, to encourage the use of these test suites, and ensure their likely use by even kernel developers, these test suites were included in the standard Frysk installation (e.g., in Fedora the `frysk-devel` RPM contains all of Frysk's test suites).

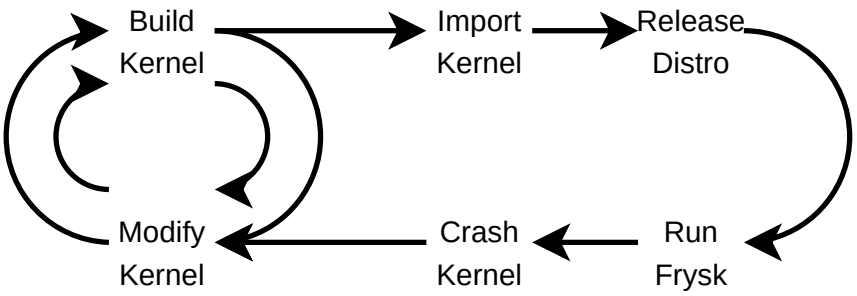


Figure 6: Long Feedback Cycle—`ptrace` interface

```
$ /usr/lib/frysk/fsystest
SKIP: frysk2595/ptrace_after_forked_thread_exits
PASS: frysk2595/ptrace_after_exec
....
PASS: frysk3525/exit47
PASS: frysk3595/detach-multi-thread
PASS: frysk2130/strace-clone-exec.sh
XFAIL: frysk2595/ptrace_peek_wrong_thread
```

Figure 7: Running Frysk's `fsystest`

Figure 7 illustrates the running of Frysk's "upstream" or system test suite using `fsystest`, and Figure 8 illustrates the running of Frysk's own test suite, implemented using the `JUnit` test framework.

6 Problems Identified by Frysk

In this final section, two specific tests included in Frysk's "upstream" test suite will be described.

6.1 Case 1: Clone-Exec Crash

In applying test-driven development, Frysk developers first enumerate, and then implement all identifiable sequences of a given scenario. For instance, to ensure that `exec` can be correctly handled, the Frysk test suite sequences many scenarios including the following:

- 32-bit program exec'ing a 64-bit program
- main thread exec'ing
- non-main thread exec'ing

The last case, as illustrated in Figure 9, has proven to be especially interesting. When first implemented,

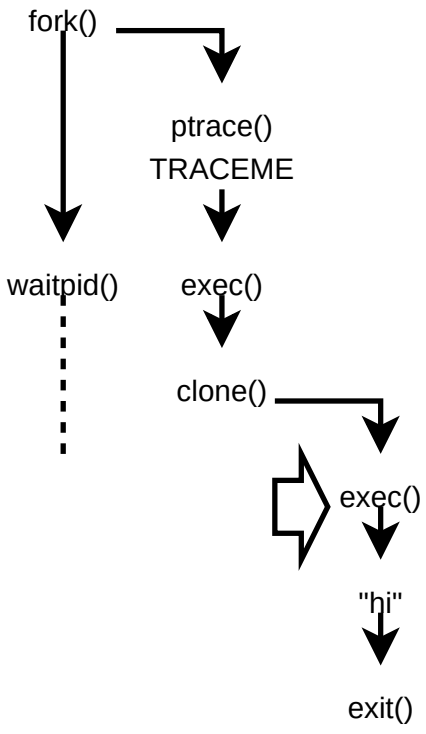


Figure 9: Clone Exec Crash

```
$ /usr/lib/frysk/funit
Running testAssertEOF(frysk.expunit.TestExpect) ...PASS
Running testTimeout(frysk.expunit.TestExpect) ...PASS
Running testEquals(frysk.expunit.TestExpect) ...PASS
Running testIA32(frysk.sys.proc.TestAuxv) ...PASS
Running testAMD64(frysk.sys.proc.TestAuxv) ...PASS
Running testIA64(frysk.sys.proc.TestAuxv) ...PASS
Running testPPC32(frysk.sys.proc.TestAuxv) ...PASS
Running testPPC64(frysk.sys.proc.TestAuxv) ...PASS
....
```

Figure 8: Running Frysk's `funit`

it was found that the 2.6.14 kernel had a regression causing Frysk's unit-test to fail—the traced program would core dump. Consequently, a standalone test `strace-clone-exec.sh` was added to Frysk's "upstream" test suite demonstrating the problem, and the fix was pushed upstream.

Then later with the availability of the 2.6.18 kernels with the `utrace` patch, it was found that running Frysk's test suite could trigger a kernel panic. This was quickly isolated down to the same `strace-clone-exec.sh` test, but this time running the test caused a kernel panic. Since the test was already widely available, a fix could soon be written and incorporated upstream.

6.2 Case 2: Threaded ptrace Calls

Graphical debugging applications, such as Frysk, are often built around two or more threads:

- an event-loop thread handling process start, stop, and other events being reported by the kernel.
- a user-interface thread that responds to user requests such as displaying the contents of a stopped process' memory, while at the same time ensuring that the graphical display remains responsive.

As illustrated in Figure 10, Linux restricts all `ptrace` requests to the thread that made the initial `PTRACE_ATTACH`. Consequently, any application using `ptrace` is forced to route all calls through a dedicated thread. In the case of Frysk, initially a dedicated `ptrace` thread was created, but later that thread's functionality was folded into the event-loop thread.

The "upstream" test `ptrace_peek_wrong_thread` was added to illustrate this kernel limitation.

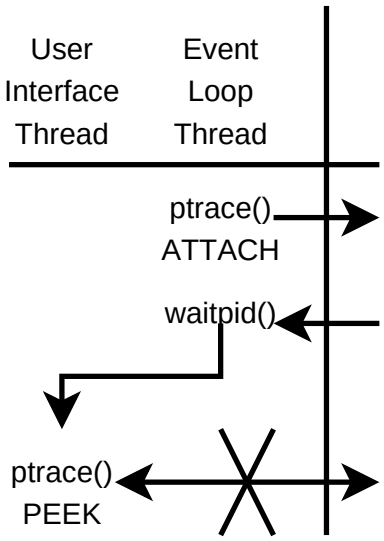


Figure 10: Multi-threaded ptrace

7 Conclusion

As illustrated by examples such as the Exec Crash bug described in Section 6.1, Frysk, by both implementing an "upstream" test suite (focused largely on the kernel) and including that test suite in a standard install, has helped to significantly reduce the lag between a kernel change affecting a debug-interface on which it depends (such as `ptrace`) and that change been detected and resolved.

And the final score? Since Frysk's stand-alone test suite is identifying limitations and problems in the existing `ptrace` and `/proc` interfaces and the problems in the new `utrace` code, this one can be called a draw.

8 Acknowledgments

Special thanks to the Toronto Windsurfing Club Hatteras Crew for allowing me to write this, and not forcing me to sail.

References

The Frysk Project, `http://sourceware.org/frysk`

Clone-exec Crash Bug, `http://sourceware.org/bugzilla/show_bug.cgi?id=2130`

Threaded ptrace Calls Bug, `http://sourceware.org/bugzilla/show_bug.cgi?id=2595`

Roland McGrath, *`utrace` Patches*, `http://people.redhat.com/roland/utrace/`

The JUnit Project, `http://www.junit.org/index.htm`

The Java-GNOME Project, `http://java-gnome.sourceforge.net/`

GCJ, `http://gcc.gnu.org/java/`

ExpUnit, `http://sourceware.org/frysk/javadoc/public/frysk/expunit/package-summary.html`

Dogtail, `http://people.redhat.com/zcerza/dogtail/`

`expect`, `http://expect.nist.gov/`

Keeping Kernel Performance from Regressions

Tim Chen
Intel Corporation
tim.c.chen@intel.com

Leonid I. Ananiev
leoan@mail.ru

Alexander V. Tikhonov
Intel Corporation
alexander.v.tikhonov@intel.com

Abstract

The Linux* kernel is evolving rapidly with thousands of patches monthly going into the base kernel. With development at this pace, we need a way to ensure that the patches merged into the mainline do not cause performance regressions.

The Linux Kernel Performance project was started in July 2005 and is Intel's effort to ensure every dot release from Linus is evaluated with key workloads. In this paper, we present our test infrastructure, test methodology, and results collected over the 2.6 kernel development cycle. We also look at examples of historical performance regressions that occurred and how Intel and the Linux community worked together to address them to make Linux a world-class enterprise operating system.

1 Introduction

In recent years, Linux has been evolving very rapidly, with patches numbering up to the thousands going into the kernel for each major release (see Figure 1) in roughly a two- to three-month cycle. The performance and scalability of the Linux kernel have been key ingredients of its success. However, with this kind of rapid evolution, changes detrimental to performance could slip in without detection until the change is in the distributions' kernels and deployed in production systems. This underscores the need for a systematic and disciplined way to characterize, test, and track Linux kernel performance, to catch any performance issue of the kernel at the earliest time possible to get it corrected.

Intel's Open Source Technology Center (OTC) launched the Linux Kernel Performance Project (LKP) in the summer of 2005 (`http://kernel-perf.sourceforge.net`) to address the need to monitor kernel performance on a regular basis. A group of OTC engineers set up the test machines, infrastructure, and benchmarks; they started regular testing and analysis of Linux kernel's performance. We began to publish our test data on project website since July 2005.

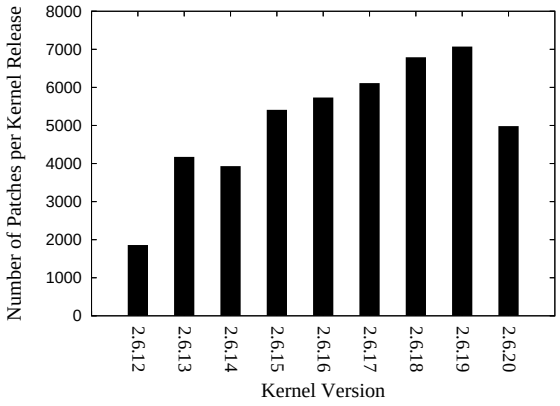


Figure 1: Rate of change in Linux kernel

2 Testing Process

Each release candidate of the Linux kernel triggers our test infrastructure, which starts running a benchmark test suite within an hour whenever a new kernel get published. Otherwise, if no new `-rc` version appears within a week, we pick the latest snapshot (`-git`) kernel for testing over the weekend. The test results are reviewed weekly. Anomalous results are double-checked, and rerun if needed. The results are uploaded to a database accessible by a web interface. If there were any significant performance changes, we would investigate the causes and discuss them on Linux kernel mailing list (see Figure 2).

We also make our data available on our website publicly for community members to review performance gains and losses with every version of the kernel. Ultimately, we hope that this data catches regressions before major kernel releases, and results in consistent performance improvement.

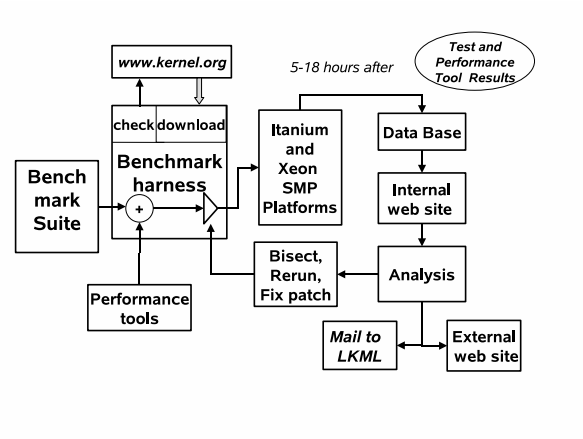


Figure 2: Performance testing process.

2.1 Benchmarks Suite

We ran a set of benchmarks covering core components of the Linux kernel (memory management, I/O subsystem, process scheduler, file system, network stack, etc.).

Table 1 lists and describes the benchmarks. Most of the benchmarks are open source and can be duplicated easily by others.

2.2 Test Platforms

Currently we have a mix of Itanium® SMP machines and Xeon® SMP machines to serve as our test platforms, with the configurations as listed below:

4P Intel® Itanium™ 2 processor (1.6Ghz)
4P Dual-Core Intel® Itanium® processor (1.5Ghz)
2P Intel® Xeon® MP processor (3.4Ghz)
4P Dual-Core Intel® Xeon® MP processor (3.0Ghz)
2P Xeon® Core™2 Duo processor (2.66Ghz)
2P Xeon® Core™2 Quad processor (2.40Ghz)

2.3 Test Harness

We needed a test harness to automate the regular execution of benchmarks on test machines. Even though there were test harnesses from the Scalable Test Platform (`http://sourceforge.net/projects/stp`) and Linux Test Project (`http://ltp.sourceforge.net`), they did not fully meet all of our testing requirements. We elected to create a set of shell scripts for our test harness, which was easy to customize for adding the capabilities we need.

Short name	**Description**
Kbuild	Measures the speed of Linux kernel compilation.
Reaim7	Stresses the scheduler with up to thousands of threads each generating load on memory and I/O.
Volanomark	A chatroom benchmark to test java thread scheduling and network scalability.
Netperf	Measures the performance of TCP/IP network stack.
Tbench	Load testing of TCP and process scheduling.
Dbench	A stress test emulating Netbench load on file system.
Tiobench	Multithread IO subsystem performance test.
Fileio	Sysbench component for file I/O workloads.
Iozone	Tests the file system I/O under different ratio of file size to system memory.
Aiostress	Tests asynchronous I/O performance on file system and block device.
Mmbench	Memory management performance benchmark.
Httperf	Measures web server performance; also measures server power usage information under specific offered load levels.
Cpu-int/fp	An industry standard CPU intensive benchmark suite on integer and floating point operations.
Java-business	An industry standard benchmark to measure server-side Java*, tests scheduler scalability.

Table 1: Performance benchmark suite

Our test harness provides a list of services that are itemized below:

- It can detect and download new Linux kernels from `kernel.org` within 30 minutes after release, and then automatically install the kernel and initiate benchmark suites on multiple test platforms.

- It can test patches on any kernel and compare results with other kernel version.
- It can locate a patch that causes a performance change by automating the cycle of git-bisect, kernel install for filtering out the relevant patch.
- It uploads results from benchmark runs for different kernels and platforms into a database. The results and corresponding profile data can be accessed with a friendly web interface.
- It can queue tasks for a test machine so that different test runs can be executed in sequence without interference.
- It can invoke a different mix of benchmarks and profiling tools.

We use a web interface to allow easy comparison of results from multiple kernels and review of profiling data. The results may be rerun using the web interface to confirm a performance change, or automated git-bisect command be initiated to locate the patch responsible. The results are published in external site (`http://kernel-perf.sourceforge.net`) after they have been reviewed.

3 Performance Changes

During the course of the project, our systematic testing has revealed performance issues in the kernels. A partial list of the performance changes are listed in Table 2. We will go over some of those in details.

3.1 Disk I/O

3.1.1 MPT Fusion Driver Bug

There was a sharp drop in disk performance for the 2.6.13 kernel (see Figure 3). Initially we thought that it was related to the change in system tick from 1000Hz to 250Hz. After further investigation, it was found that the change in Hz actually revealed a race condition bug in the MPT fusion driver's initialization code.

Our colleague Ken Chen found that there were two threads during driver initialization interfering with each other: one for domain validation, and one for host controller initialization. When there were two host controllers, while the second host controller was brought up, the initialization thread temporarily disabled the channel for the first controller. However, domain validation was in progress on first channel in another thread (and possibly running on another CPU). The effect of disabling the first channel during in-progress domain validation was that it caused all subsequent domain validation commands to fail. This resulted in the lowest possible performance setting for almost all disks pending domain validation. Ken provided a patch and corrected the problem.

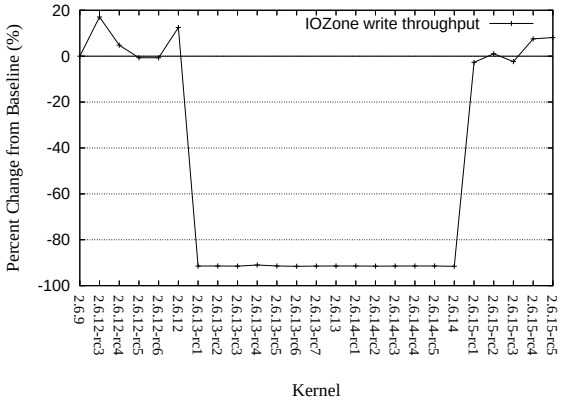


Figure 3: MPT Fusion driver bug

3.2 Scheduler

3.2.1 Missing Inter-Processor Interrupts

During the 2.6.15 time frame, there was a 60% decrease in Volanomark throughput on Itanium® test machines (see Figure 4). It was caused by a patch that caused rescheduled Inter Processor Interrupts (IPI) not to be sent from `resched_task()`, ending up delaying the rescheduling task until next timer tick, thus causing the performance regression. The problem was quickly reported and corrected by our team.

3.2.2 Scheduler Domain Corruption

During the testing of benchmark Httperf, we noticed unusual variation on the order of 80% in the response time of the web server under test for the 2.6.19 kernel. This was caused by a bug introduced when the

Kernel	Patch causing change	Effect
2.6.12-rc4	noop-iosched: kill O(N) merge scan.	Degraded IO throughput for noop IO scheduler by 32%.
2.6.13-rc2	Selectable Timer Interrupt Frequency of 100, 250, and 1000 HZ.	Degraded IO throughput by 43% due to MPT Fusion driver.
2.6.15-rc1	sched: resched and cpu_idle rework.	Degraded performance of Netperf (-98%) and Volanomark (-58%) on ia64 platforms.
2.6.15-rc2	ia64: cpu_idle performance bug fix	Fixed Volanomark and netperf degradations in 2.6.15-rc1.
2.6.15-rc5	[SCSI] mptfusion : driver performance fix.	Restored fileio throughput.
2.6.16-rc1	x86_64: Clean up copy_to/from_user. Remove optimization for old B stepping Opteron.	Degraded Netperf by 20% on Xeon® MP.
2.6.16-rc3	x86_64: Undid the earlier changes to remove unrolled copy/memset functions for Xeon® MP.	Reverted the memory copy regression in 2.6.16-rc1.
2.6.18-rc1	lockdep: irqtrace subsystem, move account_system_vtime() calls into softirq.c.	Netperf degraded by 3%.
2.6.18-rc4	Reducing local_bh_enable/disable overhead in irq trace.	Netperf performance degradation in 2.6.18-rc1 restored.
2.6.19-rc1	mm: balance dirty pages Now that we can detect writers of shared mappings, throttle them.	IOzone sequential write dropped by 55%.
2.6.19-rc1	Send acknowledge each 2nd received segment.	Volanomark benchmark throughput reduced by 10%.
2.6.19-rc1	Let WARN_ON return the condition.	Tbench degraded by 14%.
2.6.19-rc2	Fix WARN_ON regression.	Tbench performance restored.
2.6.19-rc2	elevator: move the back merging logic into the elevator core	Noop IO scheduler performance in 2.6.18-rc4 fixed and restored to 2.6.12-rc3 level

Table 2: Linux kernel performance changes seen by test suites

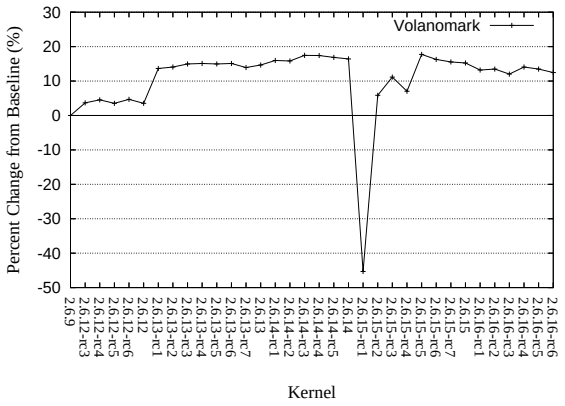


Figure 4: IPI scheduling bug

`cpu_isolated_map` structure was designated as init data. However, the structure could be accessed again after the kernel was initialized and booted when a rebuild of `sched_domain` was triggered by setting the `sched_mc_power_savings` policy. Subsequently, the corrupted `sched_domain` caused bad load-balancing behavior and caused erratic response time.

3.2.3 Rotating Staircase Dead Line Scheduler

The recently proposed RSDL (Rotating Staircase Dead Line) scheduler has generated a lot of interest due to its elegant handling of interactivity. We put RSDL 0.31 to test and found that for Volanomark, there is a big 30% to 80% slowdown. It turned out that the yield semantics in RSDL 0.31 were too quick to activate the yielding process again. RSDL 0.33 changed the yield semantics to allow other processes a chance to run, and the performance recovered.

3.3 Memory Access

3.3.1 Copy Between Kernel and User Space

During the 2.6.15-rc1 timeframe, we detected a drop up to 30% in Netperf's throughput on older Xeon® processor MP-based machines (see Figure 5). This was caused by a switch in the copy between user and kernel space to use repeat move string instructions which are slower than loop-based copy on Xeon® processor MP. This problem was corrected quickly. Later, when

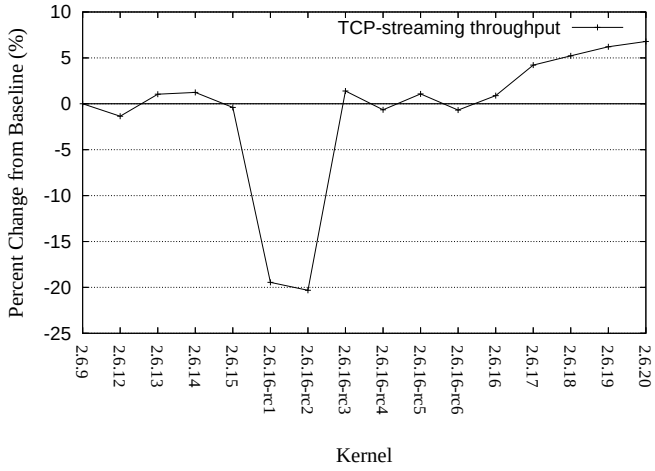


Figure 5: Xeon® processor MP's Netperf TCP-streaming performance made worse using string copy operations

the newer Core™2 Duo based Xeon®s became available with efficient repeat move string instructions, a switch to use string instructions in the 2.6.19 kernel actually greatly improved throughput. (see Figure 6).

3.4 Miscellaneous

3.4.1 Para-Virtualization

The para-virtualization option was introduced in the 2.6.20 time frame, and we detected a 3% drop in Netperf and Volanomark performance. We found that Para-virtualization has turned off VDSO, causing `int 0x80` rather than the more efficient `sysenter` to be used for system calls, causing the drop.

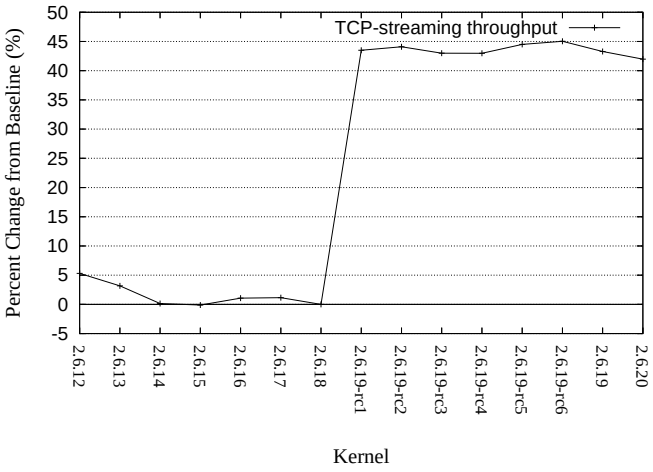


Figure 6: Xeon® Core™2 Duo's Netperf TCP-streaming performance improved with string copy operations

3.4.2 IRQ Trace Overhead

When the IRQ trace feature was introduced in 2.6.18-rc1, it unconditionally added `local_irq_save(flags)` and `local_irq_restore(flags)` when enabling/disabling bottom halves. This additional overhead caused a 3% regression in Netperf's UDP streaming tests, even when the IRQ tracing feature was unused. This problem was detected and corrected.

3.4.3 Cache Line Bouncing

There was a 16% degradation of tbench in 2.6.18-rc14 (see Figure 7) We found that a change in the code triggered an inappropriate object code optimization in older gcc 3.4.5, which turned a rare write into a global variable into an always write event to avoid a conditional jump. As a result, cache line bouncing among the cpus increased by 70% from our profiling. A patch was later merged by Andrew into the mainline to sidestep this gcc problem.

4 Test Methodology

4.1 Test Configuration

We tried to tune our workloads so they could saturate the system as much as possible. For a pure

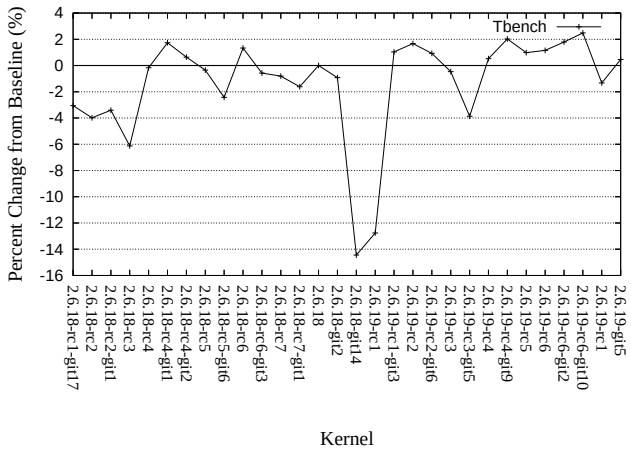


Figure 7: Tbench performance problem caused by cache line bouncing

CPU-bound workload, the CPU utilization was close to 100%. However for a workload involving I/O, the system utilization was limited by the time waiting for file and network I/O. The detailed benchmark options for each test are described on our website (`http://kernel-perf.sourceforge.net/about_tests.php`). Table 3 gives a sense of the average loading for different benchmarks. The loading profile is the standard one from vmstat.

For the disk-bound test workload, we reduced the amount of main memory booted to only 1GB (that's only a quarter to one-eighth of the memory of our system). The test file size was a few times of the size of memory booted. This made the actual effect of I/O dominant and reduced the effect of file I/O cache.

Name	% cpu	% io	% mem	% user	% sys
Reaim7	100	1	68	85	15
Aiostress	1	36	83	0	1
Dbench	37	28	95	1	36
Fileio	1	14	100	0	1
IOzone	1	23	99	0	1
Kbuild	79	9	90	74	5
Mmbench	2	66	99	0	2
Netperf	40	0	34	2	38
Cpu-int/fp	100	0	75	100	0
Java-business	39	0	89	39	0
tbench	97	0	41	5	92
Volanomark	99	0	96	45	54

Table 3: Sample system loading under benchmarks

4.2 Dealing with Test Variation

Variations in performance measurements are part of any experiment. To some extent the starting state of the system, like cpu cache, file I/O cache, TLB, and disk geometry, played a role. However, a large variation makes the detection of change in performance difficult.

To minimize variation, we do the following:

- Start tests with a known system state;
- Utilize a warm-up workload to bring the system to a steady state;
- Use a long run time and run the benchmark multiple times to get averages of performance where possible.

To get our system in known state, we rebooted our system before our test runs. We also reformatted the disk and installed the test files. This helped to ensure the layout of the test file and the location of journal on the disk to remain the same for each run.

We also ran warm-up workloads before the actual benchmark run. This helped bring the CPU caches, TLB, and file I/O cache into a steady state before the actual testing.

The third approach we took was to either run the benchmark for a long time or to repeat the benchmark run multiple times and measure the average performance. Short tests like Kbuild, when run repeatedly for 15 times in our test run, got a good average value with standard deviation below 1%. The average performance value has reduced variance and resembles more closely a well behaved Gaussian distribution [5]. Single run results for some benchmarks are far from a Gaussian distribution. One such example is Tbench.

Figure 8 superimposes the distribution of throughput from a single run of Tbench versus a normal distribution with the same standard deviation. It can be seen that the distribution from a single benchmark run is bimodal and asymmetric. Therefore using a single measurement for comparison is problematic with the issues raised in [1-4]. For these cases, a more accurate performance comparison is made by using average values, which resemble much more closely a normal distribution and have smaller deviations.

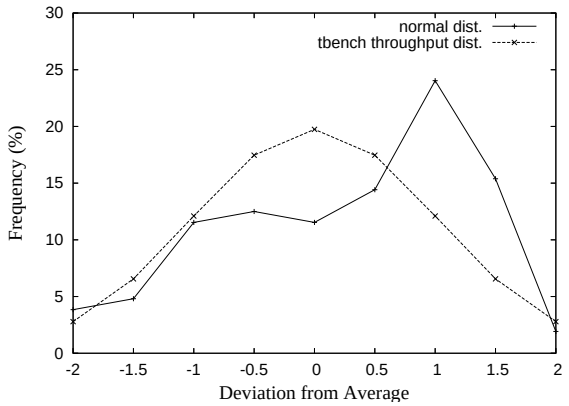


Figure 8: Tbench throughput distribution

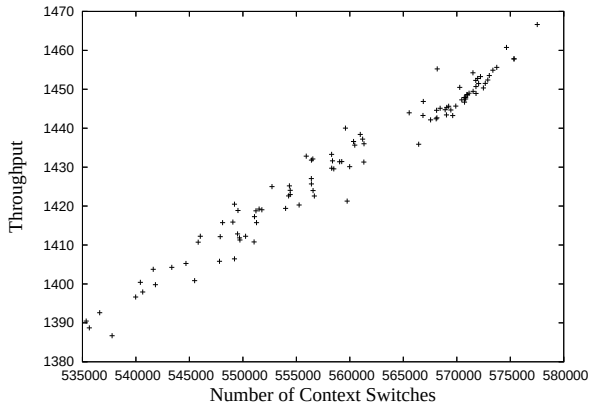


Figure 9: Tbench throughput vs. context switches

Sometimes we can better understand the underlying reason for performance variation by correlating the performance variation with changes in other profiling data. For example, with Tbench, context switch has a 0.98 correlation coefficient with the throughput (see Figure 9). This gives an indication that the variation in context switch rate is highly correlated with the variation in throughput. Another example is Kbuild (see Figure 10), where we find the number of merged IO blocks had a –0.96 correlation coefficient with the kernel compile time, showing that the efficiency of disk I/O operations in merging IO blocks is critical to throughput.

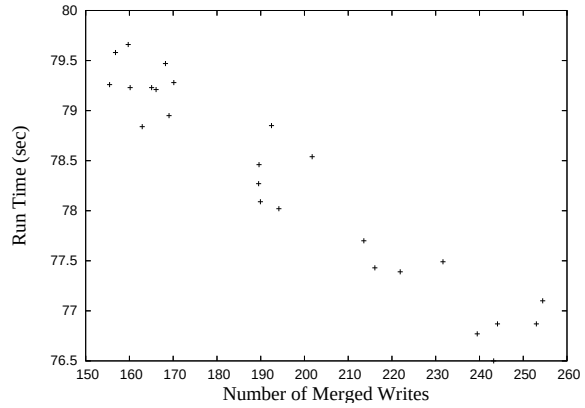


Figure 10: Kbuild runtime vs. number of merged IO blocks

This kind of almost one-to-one correlation between throughput and profiling data can be a big help to check whether there is a real change in system behavior. Even though there are variations in throughput from each run, the ratio between the throughput and profile data should be stable. So when comparing two kernels, if there is a significant change in this ratio, we will know that there are significant changes in the system behavior.

We have also performed a large number of runs of benchmarks on a baseline kernel to establish the benchmark variation value. Both max and min values are saved in a database to establish a confidence interval for a benchmark. This value is used for results comparison: if the difference in measured performance values is more than the confidence interval, then there is a significant change in the kernel performance that should be looked into. In general, disk-I/O-bound benchmarks have much higher variation, making it much harder to detect small changes in performance in them.

4.3 Profiling

Our test harness collected profiling data during benchmark runs with a set of profiling tools: `vmstat`, `iostat`, `sar`, `mpstat`, `ps`, and `readprofile`. The profiling data provided information about the load on the CPU from user applications and the activities of the kernel's subsystems: scheduler, memory, file, network, and I/O. Information about I/O queue length and hot kernel functions had been useful for us in locating bottlenecks in the kernel and to investigate the cause of performance changes. The waiting time from `vmstat` can be combined with `wchan` information from `ps` to gain insight to time spent by processes waiting for events. Table 4 provides a profile of waited events for a run snapshot of Aiostress and Reaim7 benchmarks as an example.

Aiostress		Reaim7	
1209	pause	18075	pause
353	io_getevents	8120	wait
13	get_write_access	1218	exit
12	sync_buffer	102	pipe_wait
6	stext	62	start_this_handle
3	sync_page	1	sync_page
1	congestion_wait	2	sync_page
1	get_request_wait	2	cond_resched

Table 4: The events waited by Aiostress and Reaim7

4.4 Automated Git-Bisect

The git bisect utility is a powerful tool to locate the patch responsible for a change in behavior of the kernel from a set of patches. However, manually running it to bisect a large patch-set repeatedly to find a patch is tedious. One has to perform the steps of bisecting the patch set into two, rebuild, install, and reboot the kernel for one patch set, run the benchmark to determine if the patch set causes an improvement or degradation to the performance, and determine which subset of the two bisected patch sets contains the patch responsible for the change. Then the process repeats again on a smaller patch set containing the culprit. The number of patches between two rc releases are in the hundreds, and often 8 to 10 repetitions are needed. We added capability in our test harness to automate the bisect process and benchmark run. It is a very useful tool to automatically locate the patch responsible for any performance change in O(log n) iterations.

4.5 Results Presentation

After our benchmark runs have been completed, a wrapper script collects the output from each benchmark and puts it into a 'comma separated value' format file that is parsed into a MySQL* database. The results are accessible through an external web site `http://kernel-perf.sourceforge.net` as a table and chart of percentage change of the performance compared to a baseline kernel (selected to be 2.6.9 for older machines, and 2.6.18 for newer ones). Our internal web site shows additional runtime data, kernel config file, profile results, and a control to trigger a re-run or to perform a git bisect.

4.6 Performance Index

It is useful to have a single performance index that summarizes the large set of results from all the benchmarks being run. This approach has been advocated in the literature (see [1]-[4]). This is analogous to a stock market index, which gives a sense of the overall market trend from the perspective of individual stock, each weighted according to a pre-determined criterion.

Benchmark	Number of subtests	Deviation %	Weight per metric
Reaim7	1	0.46	2
Aiostress	8	12.8	0.01
Cpu-int/fp	2	0.6	1
Dbench	1	11.3	0.1
fileio	1	11.8	0.1
Iozone	21	14.7	0.01
Kbuild	1	1.4	1
Mmbench	1	4.9	0.2
Netperf	7	1.6	0.15
Java-Business	1	0.6	1
tbench	1	12.7	0.5
Tiobench	9	11.4	0.01
Volanomark	1	0.8	1

Table 5: Number of subtests, variations weights on subtests for each benchmark

We use the geometric mean of ratios of performance metric to its baseline value of each benchmark as a performance index, as suggested in [2]. We weigh each benchmark according to its reliability (i.e., benchmarks with less variance are weighed more heavily). If a benchmark has a large number of subtests producing multiple metrics, we put less weight on each metric so the benchmark will not be over-represented in the performance index. Table 5 shows the weights being chosen for some of our benchmarks.

We use a weighted version of the geometric mean to aid us in summarizing the performance of the kernel. This weighted geometric index, though somewhat subjective, is a very useful tool to monitor any change in overall kernel performance at a glance and help guide us to the specific benchmarked component causing the change. Figure 11 shows the performance index produced over our benchmark suite. It is interesting to note that from the limited perspective of the benchmark suite

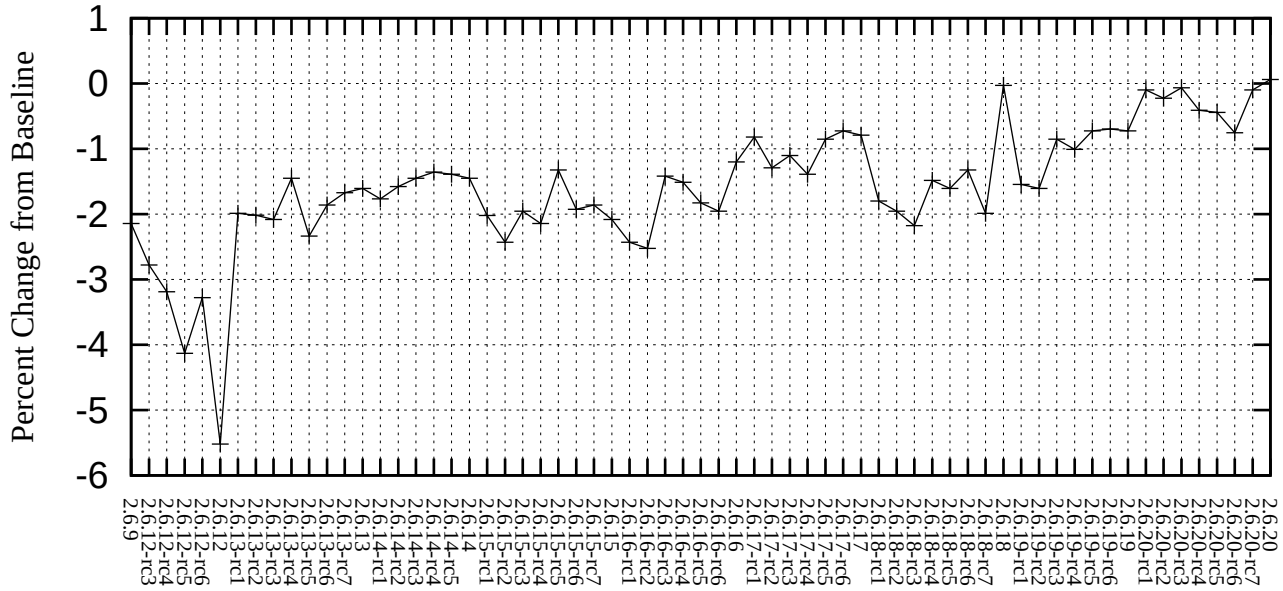


Figure 11: Weighted geometric mean performance for all benchmarks

we run regularly, the index for the 2.6 kernel series has been trending upwards.

5 Conclusion

Our project set up the infrastructure to systematically test every kernel release candidate across multiple platforms and benchmarks, and also made the test data available to the community on the project website, `http://kernel-perf.sourceforge.net`. As a result, we have been able to catch some kernel regressions quickly, and worked with the community to fix them. However, with rapid changes in the kernel, the limited coverage from our regular benchmark runs could uncover only a portion of performance regressions. We hope this work will encourage more people to do regular and systematic testing of the Linux kernel, and help prevent performance problems from propagating downstream into distribution kernels. This will help to solidify Linux's position as a world-class enterprise system.

Acknowledgments

The authors are grateful to our previous colleagues Ken Chen, Rohit Seth, Vladimir Sheviakov, Davis Hart, and Ben LaHaise, who were instrumental in the creation of the project and its execution. A special thanks to Ken, who was a primary motivator and contributor to the project.

Legal

References

[1] J.E. Smith, "Characterizing Computer Performance with a Single Number," Communications of ACM, 31(10):1202–1206, October 1988.

[2] J.R. Mashey, "War of the Benchmark Means: Time for a Truce" ACM SIGARCH Computer Architecture News. Volume 32, Issue 4 (September 2004), pp. 1–14. ACM Press, New York, NY, USA.

[3] J.R. Mashey, "Summarizing Performance is No Mean Feat" Workload Characterization Symposium, 2005. Proceedings of the IEEE International. 6–8 Oct. 2005 Page(s): 1. Digital Object Identifier 10.1109/IISWC.2005.1525995

[4] L. John, "More on finding a Single Number to indicate Overall Performance of a Benchmark Suite," Computer Architecture News, Vol. 32, No 1, pp. 3–8, March 2004.

[5] D.J. Lilja, "Measuring computer performance: a practitioner's guide," Cambridge University Press, 2005.

Breaking the Chains—Using LinuxBIOS to Liberate Embedded x86 Processors

Jordan H. Crouse
Advanced Micro Devices, Inc.
jordan.crouse@amd.com

Marc E. Jones
Advanced Micro Devices, Inc.
marc.jones@amd.com

Ronald G. Minnich
Sandia National Labs

Abstract

While x86 processors are an attractive option for embedded designs, many embedded developers avoid them because x86-based systems remain dependent on a legacy BIOS ROM to set up the system. LinuxBIOS is an open source solution that replaces the proprietary BIOS ROMs with a light-weight loader. LinuxBIOS frees the developer from complex CPU and chipset initialization and allows a variety of payloads to be loaded, including the Linux kernel itself.

This presentation reviews the journey of the AMD Geode™ processors and chipset as they were integrated into LinuxBIOS to become the centerpoint of the One Laptop Per Child (OLPC) project. We also discuss how embedded developers can take advantage of the LinuxBIOS environment for their own x86-based projects.

1 Introduction

Ever since the x86 Personal Computer (PC) architecture was introduced in 1981, it has been accompanied by bootstrap code known as the Basic Input/Output System (BIOS) that executes a Power On Self Test (POST). Almost every year since the PC's introduction, hardware and operating system features have increased in complexity. Each new feature adds complexity to the BIOS, which must maintain compatibility with older operating systems and yet also provide support for new ones. The end result is a convoluted and cryptic combination of old standards (such as software interrupts for accessing the display and storage devices) and new standards (such as Advanced Configuration and Power Interface (ACPI)).

Almost all BIOS implementations are proprietary and many Open Source developers are in conflict with what is perceived to generally be a "black magic" box. Due to the arcane nature of the BIOS, most modern operating systems have abandoned the BIOS hardware interfaces and access the hardware directly. The desktop computer focus of the traditional BIOS model frustrates embedded systems designers and developers, who struggle to get a BIOS that embraces their unique platforms. Due to the very specific requirements for system boot time and resource usage, it is difficult to meet embedded standards with a BIOS designed for two decades of desktop computers.

The LinuxBIOS project exists to address legacy BIOS issues. It is licenced under the GNU Public License (GPL) to promote a transparent and open loader. LinuxBIOS provides CPU and chipset initialization for x86, x86_64, and Alpha systems and allows the flexibility to load and run any number of different payloads.

This paper discusses the development and use of LinuxBIOS for embedded x86 platforms based on AMD Geode processors. The first section examines the history of LinuxBIOS and the AMD Geode processors. The next section moves into detail about the system initialization process. The final section discusses integrating payloads with the LinuxBIOS firmware.

2 History

> "History is a guide to navigation in perilous times. History is who we are and why we are the way we are."
>
> *—David C. McCullough*

2.1 LinuxBIOS History

Ron Minnich started the LinuxBIOS project at Los Alamos National Lab (LANL) in September 1999 to address problems caused by the PC BIOS in large clusters. The team agreed that the ideal PC cluster node would have the following features:

- Boot directly into an OS from non-volatile RAM;
- Configure only the network interfaces;
- Connect to a control node using any working network interface;
- Take action only at the direction of the control node.

At that time, the LANL team felt that Linux® did a better job of running the hardware than the PC BIOS. Their concept was to use a simple hardware bootstrap to load a small Linux kernel from flash to memory. Leveraging work from the OpenBIOS project, the LinuxBIOS team booted an Intel L440GX+ motherboard after approximately six months of development. Early on, the team decided that assembly code would not be the future of LinuxBIOS. OpenBIOS was disregarded because it was based on a great deal of assembly code and a difficult-to-master build structure. The team found a simple loader from STMicroelectronics called STPC BIOS that was written in C and available to be open sourced, so it became the basis for the first version of LinuxBIOS.[1]

In 2000, Linux NetworX and Linux Labs joined the effort. The LinuxBIOS team added Symmetric Multiple Processor (SMP) support, an Alpha port, and created the first 13-node LinuxBIOS-based Supercomputing Clusters. Since 2001, the team has added developers and they continue to port to new platforms, including AMD Opteron™ processor- and AMD Athlon™ processor-based platforms. Interestingly enough, LinuxBIOS was originally designed for clusters, yet LinuxBIOS for non-cluster platforms far exceeds the cluster use.

In 2005, some current and past members of the MIT Media Lab joined together to create the One Laptop Per Child (OLPC) program, dedicated to making a low-cost laptop for educational projects around the globe. The OLPC team designed an x86 platform that incorporates an AMD Geode solution. As low price and open technology were part of the core requirements for the laptop, the designers decided to use a royalty-free open source BIOS solution, ultimately choosing LinuxBIOS. The first two board revisions included the AMD Geode GX processor based LinuxBIOS loader, originally utilizing a Linux-as-bootloader payload. This later transitioned to OpenFirmware after it became available in the middle of 2006. In 2007, AMD Geode LX processor support was added to the loader, making it a freely available reference BIOS design for interested developers of other AMD Geode solutions.

[1] Version 2 started after the addition of Hypertransport™ technology support changed the device model enough to warrant a version bump.

2.2 AMD Geode History

The AMD Geode processor is the offspring of the MediaGX processor released by Cyrix in 1997. The MediaGX saved total system cost by embedding a graphics engine that used one of the first Unified Memory Architecture (UMA) implementations. It also featured an integrated northbridge memory controller and SoundBlaster emulation in the CPU. The MediaGX broke the sub-$1000, sub-$500, and sub-$200 price barrier on the Compaq Presario 2100 in 1996 and 1997. In 1997, Cyrix was purchased by National Semiconductor, who renamed the MediaGX line to Geode. National Semiconductor released the Geode GX2 (today, just called the GX) and CS5535 companion chip in 2002. In 2003, the Geode division was sold to AMD. AMD focused heavily on power, price, and performance, and in 2005 released the AMD Geode LX 800@0.8W processor and CS5536 companion chip, with the LX 900@1.5W processor following in 2007.

The AMD Geode GX and LX processors support the i586 instruction set, along with MMX and 3DNow!™ extensions. The LX features a 64K instruction and a 64K data L1 cache and 128K L2 cache. Both processors have on-board 2D graphics and video accelerators. The LX adds an on-board AES engine and true random number generator. The CS5536 companion chip provides southbridge capabilities: IDE, USB 2.0, SMBus, AC97, timers, GPIO pins, and legacy x86 peripherals.

3 Geode LinuxBIOS ROM image

While the entire image is known as LinuxBIOS, it is constructed of individual pieces that work together. An AMD Geode LinuxBIOS ROM image is made up of three main binary pieces:[2]

- LinuxBIOS Loader: system initialization code;

[2] OLPC also adds additional binary code for its embedded controller.

- VSA2: the AMD Geode proccessor's System Management Interrupt (SMI) handler;
- Payload: the image or program to be loaded to boot the OS.

3.1 LinuxBIOS Architecture

LinuxBIOS version 2 is structured to support multiple motherboards, CPUs, and chipsets. The overall platform configuration is described in `Config.lb` in the `mainboard` directory. The `Config.lb` file contains important information like what CPU architecture to build for, what PCI devices and slots are present, and where code should be addressed. The mainboard also contains the pre-DRAM initialization file, `auto.c`. ROMCC compiles `auto.c` and generates a stackless assembly code file, `auto.inc`. The use of ROMCC works well for small sections of simple C code, but for complicated memory controller initialization, there are some issues with code size and C variable-to-register space conversion.

To work around the ROMCC issues, Yinghai Lu of AMD developed support for the AMD64 architecture's Cache-as-RAM (CAR) feature [1]. Compiled C code makes heavy use of the stack. Only a few lines of assembly code are needed to set up the CPU cache controller to be used as temporary storage for the stack. All the pre-DRAM initialization (including memory initialization) is compiled as normal C code. Once the memory controller is configured the stack is copied to real memory and the cache can be configured as normal. The AMD Geode processors are one of two CPUs to use a CAR implementation in LinuxBIOS version 2.[3]

3.2 LinuxBIOS Directory Structure

The LinuxBIOS source tree can be a bit daunting to a newcomer. The following is a short tour of the LinuxBIOS directory structure, highlighting the parts interesting to a systems developer.

The `cpu/` directory contains the initialization code for VSA2 and the AMD Geode graphics device.

[3] See the *Future Enhancements* section for more details about CAR in LinuxBIOS version 3.

```
linuxbios/src
|-- cpu
    |-- amd
        |-- model_gx2
        |-- model_lx
```

The `mainboard/` directory contains platform-specific configuration and code. The platform `Config.lb` file contains the PCI device configuration and IRQ routing. This directory also contains the source file compiled by ROMCC.

```
linuxbios/src
|-- mainboard
    |-- amd
        |-- norwich
    |-- olpc
        |-- rev_a
```

(**Note**: 'Norwich' is the code name for an AMD Geode development platform).

The source code in `northbridge/` includes memory initialization and the PCI bridge 0 configuration and initialization. In the AMD Geode processor's architecture, the northbridge is integrated into the CPU, so the directory name is the same as the CPU.

```
linuxbios/src
|-- northbridge
    |-- amd
        |-- gx2
        |-- lx
```

The `southbridge/` directory contains the source for SMBus, flash, UART, and other southbridge device configuration and initialization.

```
linuxbios/src
|-- southbridge
    |-- amd
        |-- cs5536
```

The `target/` directory contains the platform build directories. These include the configuration files that specify the build features including ROM size, VSA2

binary size, and the desired payload binary. This is also where the ROM image is built. A script called `buildtarget` in the `linuxbios/target` directory parses the target configuration files and builds the Makefiles for the ROM image in the platform target directory.

```
linuxbios/targets
|-- amd
    |-- norwich
|-- olpc
    |-- rev_a
```

3.3 LinuxBIOS Boot Process

Figure 1 details the process of booting a system with LinuxBIOS.

1. The AMD Geode processor fetches code from the reset vector and starts executing noncompressed (pre-DRAM) LinuxBIOS from the flash ROM. The early CPU, northbridge, and southbridge initialization takes place. Once the memory is initialized, LinuxBIOS decompresses and copies the rest of itself to low memory.

2. LinuxBIOS continues system initialization by walking the PCI tree starting at bus 0. Most of the AMD Geode device's internal configuration and initialization happens at this stage. Cache, system memory, and PCI region properties are configured. The VSA2 code is decompressed into low memory and executed.

3. The VSA2 initialization makes adjustments to the UMA for graphics memory and itself. VSA2 is then copied to the top of RAM and located logically in mid-PCI memory space, 0x80000000. VSA2 initializes the runtime components. Upon completion, control is returned to LinuxBIOS.

4. LinuxBIOS finishes enumeration and initialization of all the PCI devices in the system. PCI devices are allocated memory and I/O (including the VSA2 virtualized headers) and then enabled. During the southbridge PCI configuration, the presence of IDE-versus-flash capability and other configurations not controlled directly in PCI space are set up. The CPU device is the last device to enumerate and an end-of-POST SMI is generated to signal to VSA2 that the system is configured.

5. The last stage of LinuxBIOS is to load the payload. LinuxBIOS copies itself to the top of system memory and then locates and decompresses the payload image into memory. Finally, the payload is executed. See Section 4 for more details about the payload.

3.4 VSA2

Virtual System Architecture (VSA) is the AMD Geode device's System Management Mode (SMM) software. VSA2 is the second generation of VSA that supports GX and LX CPUs and the CS5535 and CS5536 chipsets. In a traditional BIOS, VSA2 handles normal SMI/SMM tasks like bug fixes, legacy USB, and power management (legacy, APM, and ACPI). VSA2 also handles virtual PCI configuration space for the AMD Geode device's internal controllers; graphics, IDE, flash, etc. PCI virtualization translates PCI configuration-space access to the internal device's GeodeLink™ Model-Specific Registers (MSRs). PCI configuration access is infrequent and virtualization is a good way to save silicon real-estate with software.

Since Linux manages most hardware devices on its own, it only requires VSA2 PCI virtualization.

Linux kernel drivers handle the power management, USB, and graphic controllers that would normally be controlled by VSA2 in a legacy BIOS environment. In the embedded Linux environment, only the PCI virtualization portion of VSA2 is required. Omitting the unneeded code saves space in the LinuxBIOS ROM image for larger payloads. VSA2 is in the process of being ported to GNU tools and will be released as open source. This will enable the open source community to write Virtual System Modules (VSMs) for additional features or to replace VSA2 entirely with a new AMD Geode chipset SMI handler.

The VSA2 image is compressed with NRV2B and concatenated to the beginning of the LinuxBIOS (with payload) image.[4]

4 Payloads

Once LinuxBIOS provides CPU and chipset initialization for the platform, it passes control to a payload that

[4] VSA2 is added at the beginning because execution starts at the end of the ROM image, where LinuxBIOS is located.

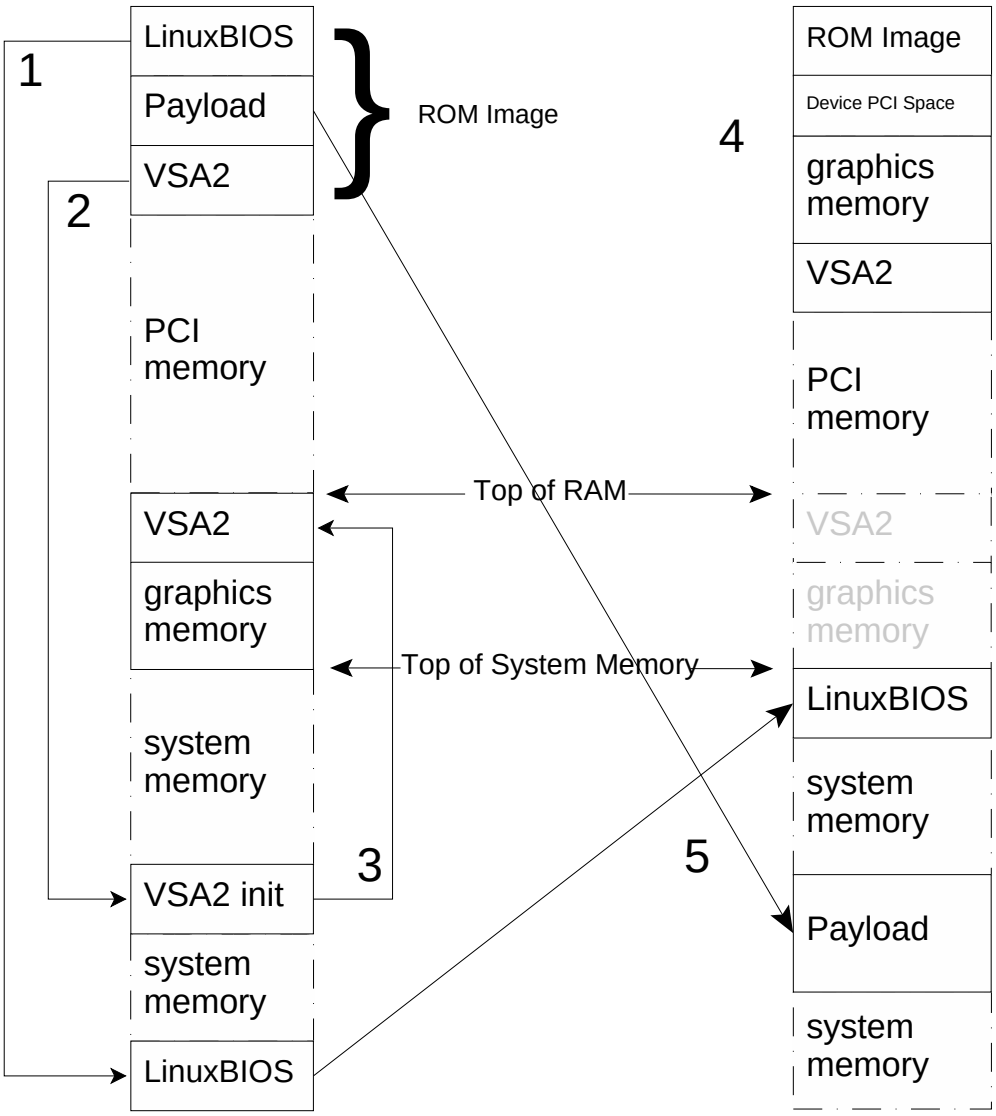


Figure 1: LinuxBIOS Memory Map

can continue the booting process. This is analogous to a traditional BIOS ROM, which also handles CPU and chipset initialization, and then passes control to code that manages the BIOS services (such as the setup screen, access to block devices, and ultimately starting the process that ends up in the secondary bootloader). The traditional BIOS code is tied closely to the loader and only provides support for a limited set of features. By contrast, a LinuxBIOS payload is far more flexible. In theory LinuxBIOS can load and run any correctly formatted ELF file (though in practice, the payload must be able to run autonomously without any operating system services). This allows the developer to choose from any number of available open source options, from simple loaders to the Linux kernel itself. This flexibility also allows embedded developers to easily craft custom solutions for their unique platform—for instance, supporting diskless boot with Etherboot, or loading and running a kernel from NAND flash or other non-traditional media.

When LinuxBIOS has finished initializing and enumerating the system, it passes control to the ELF loader to load the payload. The payload loader locates the stream on the ROM and decodes the ELF header to determine where the segments should be copied into memory. Before copying, the loader first moves the firmware to the very top of system memory to lessen the chance that it will be overwritten by the payload. LinuxBIOS stays

resident in case the ELF loader fails to load the specified payload. Many "standard" payloads (such as memtest86 and the Linux kernel) are designed to run on a system with a traditional BIOS image. Those payloads are loaded into firmware-friendly memory locations such as 0x100000. After copying and formatting the segments, the loader passes control to the entry point specified in the ELF header. System control leaves LinuxBIOS and passes to the payload.

4.1 Linux Kernel Payloads

While there are many different types of loaders, loading the Linux kernel directly from the ROM was the original goal of the LinuxBIOS project. The kernel can either be loaded by itself and mount a local or network-based filesystem, or it can be accompanied by a small RAMdisk filesystem that provides additional services for finding and booting the final kernel image. This is known as "Linux as Bootloader" or simply, LAB.

The Linux kernel is a very compelling payload for several reasons. The kernel already supports a huge number of different devices and protocols, supporting virtually any platform or system scenario. The kernel is also a well known and well supported entity, so it is easy to integrate and extend. Finally, the great majority of LinuxBIOS implementations are booting the Linux kernel anyway, so including it in the ROM greatly simplifies and accelerates the boot process. Using Linux as a bootloader further extends the flexibility by including a RAMdisk with user-space applications that can access a network or provide graphical boot menus and debug capabilities.[5]

The challenge to using a main kernel in a LinuxBIOS payload is that it is often difficult to shrink the size of the kernel to fit in the ROM. This can be mitigated by using a larger ROM. In most cases the additional cost of the flash ROM is offset by the improved security and convenience of having the main kernel in the ROM image. Another concern is the ability to safely and quickly upgrade the kernel in the ROM image. It is a dangerous matter to flash the ROM, since a failed attempt usually results in a "brick" (an unbootable machine). This can be avoided in part by increasing the size of the flash ROM and providing a safe "fallback" image that gets invoked in case of a badly flashed image. The advantages outweigh the costs for embedded applications that rarely upgrade the kernel image.

As may be expected, the standard Linux binary files require some manipulation before they can be loaded. A tool called `mkelfimage`[6] is used to combine the kernel text and data segments and to add setup code and an optional RAMdisk into a single loadable ELF file.

Table 1 shows the program headers read by `readelf` from the loadable ELF file created by `mkelfimage` from a vmlinux file and a 1MB RAMdisk.

The first segment contains code similar to the Linux startup code that de-compresses the kernel and prepares the system to boot. This section also contains setup information such as the kernel command line string. The next segment allocates space for a GDT table that is used by the setup code. Kernel developers will note the familiar `.text` segment loaded to 0x100000 and the subsequent `.data` segment. Finally, the 1MB RAMdisk is copied to address 0x800000.

4.2 Other Payloads

Several popular Open Source Software (OSS) standalone applications have been adapted to run as LinuxBIOS payloads. These include the `memtest86` memory tester and the `etherboot` network boot utility. `etherboot` is particularly interesting since it provides an open source alternative to the PXE protocol. It can easily enable any system to boot an image from a network even with network cards that do not natively support PXE boot. Another interesting option appeared during the early days of the OLPC project when Sun Microsystems unexpectedly released key portions of the OpenFirmware loader. Also known as OpenBoot, OpenFirmware is a firmware package programmed in Forth that serves as the bootloader on SPARC-based workstations and PowerPC-based systems from Apple and IBM. When it became available, OpenFirmware was quickly adapted to load on the OLPC platform as a LinuxBIOS payload.

4.3 Building Payloads

The payload is integrated with the LinuxBIOS loader during the LinuxBIOS build process. During configura-

[5] Some LinuxBIOS developers have been experimenting with fitting an entire root filesystem into the the ROM. See reference [2].

[6] Written by Eric Biderman, Joshua Aune, Jake Page, and Andrew Ip.

```
Type   Offset   VirtAddr   PhysAddr   FileSiz  MemSiz

LOAD   0x000144 0x00010000 0x00010000 0x0561c  0x1ab24
LOAD   0x005760 0x00091000 0x00091000 0x00000  0x00070
LOAD   0x005760 0xc0100000 0x00100000 0x264018 0x264018
LOAD   0x269778 0xc0365000 0x00365000 0x4b086  0xaf000
LOAD   0x2b47fe 0x00800000 0x00800000 0x100000 0x100000
```

Table 1: ELF Sections from Loadable Kernel

```
# option CONFIG_COMPRESSED_ROM_STREAM_NRV2B=0
option CONFIG_COMPRESSED_ROM_STREAM_LZMA=1
option CONFIG_PRECOMPRESSED_ROM_STREAM=1
# Need room for VSA
option ROM_SIZE=(1024*1024)-(64*1024)
...
romimage "fallback"
      ...
      payload /tmp/payload.elf
end
```

Figure 2: OLPC LinuxBIOS Configuration

tion, the developer specifies the size of the LinuxBIOS ROM image and a pointer to the payload binary. Optionally, the payload can be NRV2B- or LZMA-compressed to conserve space at the expense of speed. Figure 2 shows a set of example configuration options for an AMD Geode processor-based target with a 1MB flash ROM and a compressed payload.

During the LinuxBIOS build, the payload is compressed (if so configured), and integrated in the final ROM image as shown previously in Figure 1.

4.4 BuildROM

Constructing a LinuxBIOS ROM image from start to finish can be a complicated and tedious process involving a number of different packages and tools. BuildROM is a series of makefiles and scripts that simplify the process of building a ROM image by consolidating tasks into a single `make` target. This provides a reproducible build that can be replicated as required. BuildROM was inspired by Buildroot,[7] and was originally designed to build Linux-as-bootloader (LAB) based ROM images for the OLPC project. The OLPC LAB used a simple RAM filesystem that was based on `uClibc` and `Busybox`, and ran a simple graphical tool that could use `kexec` to load a kernel from USB, NAND, or from the network. This involved no less than six packages and a number of tools—a nightmare for the release manager, and very difficult for the core team to duplicate and run on their own platforms. BuildROM simplified the entire process and makes it easy to build new ROM image releases as they are required. More recently, it has been extended to build a number of different platforms and payloads.

[7] `http://buildroot.busybox.org`

5 Advantages and Disadvantages of LinuxBIOS

Like most open source projects, LinuxBIOS continues to be a work in progress, with both positive and negative aspects.

Chief among the positive aspects is that LinuxBIOS is developer-friendly, especially when compared to traditional BIOS solutions. LinuxBIOS is mostly C-based, which greatly simplifies development. However, machine-generated code is almost always larger and slower than hand-tuned assembly, which is a liability, especially in the pre-DRAM section where speed and size are of the essence. As mentioned before, ROMCC does an amazing job of generating stackless assembly

code, but due to the complexity of its task, it is difficult to optimize the code for minimum size and maximum efficiency.

Even though the LinuxBIOS code is written in C, the developer is not freed from having to look through the generated assembly to verify and debug the solution. Assembly code in LinuxBIOS is written in the AT&T format (as are all GNU tools-based projects), but many traditional BIOS projects and x86 CPU debuggers use the Intel format. This may cause a learning barrier for developers transitioning to LinuxBIOS, as well as making it somewhat difficult to port existing source code to LinuxBIOS.

The current AMD Geode LinuxBIOS implementation is slower then expected. Benchmarks show that decompression and memory copy are slower then other ROM implementations. More investigation is needed to determine why this happens.

The positive aspects of LinuxBIOS more than make up for these minor issues. LinuxBIOS uses a development environment familiar to embedded Linux developers. It is written in C and uses 32-bit flat mode. There is no need to worry about dealing with 16-bit real or big real modes.

In the end, while LinuxBIOS is backed by a strong open source community, it cannot exist without the support of the hardware vendors. The growth of LinuxBIOS will ultimately depend on convincing hardware companies that there is a strong business case for developing and supporting LinuxBIOS ports for their platforms.

6 Future Enhancements

There is still much to be done for the AMD Geode chipset LinuxBIOS project. LinuxBIOS version 3 promises to to be a great step forward. Among the changes planned include:

- A new configuration system based on the the kernel config system;
- Replacing remaining stackless pre-DRAM code with Cache-as-RAM (CAR) implementations;
- Speed and size optimizations in all facets of the boot process.

The AMD Geode chipset code will be transitioned to work with LinuxBIOS version 3, including better integration with the default CAR mode, and speed optimizations. Also, more work needs to be done to support a fallback image to reduce the chance that a failed ROM flash will break the target machine.

Changes are also in store for VSA2. The code will be ported to compile with GNU tools, and fully released so that others can build on the existing SMI framework. Further VSA2 work will center around power management, which will be new ground for LinuxBIOS-based ROMs. Finally, continuing work will occur to enhance BuildROM and help make more diagnostic tools available to validate and verify LinuxBIOS in an open source environment.

7 Conclusion

LinuxBIOS is an exciting development in the world of the AMD Geode chipsets and x86 platforms in general. It facilitates the efforts of developers by avoiding the pitfalls of a traditional BIOS and provides great flexibility in the unique scenarios of embedded development. There is a great advantage for the AMD Geode processors in supporting LinuxBIOS because LinuxBIOS allows designers to consider AMD Geode solutions in ways they never before thought possible (as evidenced by the early success story of the very non-traditional OLPC platform). We look forward to continuing to participate with LinuxBIOS as it transitions into version 3 and beyond.

8 Acknowledgements

The authors would like to extend a special thank you to all the people who helped make the AMD Geode LinuxBIOS image possible: Ollie Lo and the LinuxBIOS team, Yinghai Lu, Tom Sylla, Steve Goodrich, Tim Perley and the entire AMD Embedded Computing Solutions team, and the One Laptop Per Child core software team.

9 Legal Statement

References

[1] Yinghai Lu, Li-Ta Lo, Gregory Watson, Ronald Minnich. *CAR: Using Cache as RAM in LinuxBIOS*. http://linuxbios.org/data/yhlu/cache_as_ram_lb_09142006.pdf

[2] LinuxBIOS with X Server Inside, posted to *LinuxBIOS Developers Mailing List*, March 2007. http://www.openbios.org/pipermail/linuxbios/2007-March/018817.html

[3] Ronald Minnich. *LinuxBIOS at Four*, In *Linux Journal* #118, February 2004. http://www.linuxjournal.com/article/7170

[4] Ronald Minnich, *Porting LinuxBIOS to the AMD SC520*, in *Linux Journal* #136, August 2005. http://www.linuxjournal.com/article/8120

[5] Ronald Minnich, *Porting LinuxBIOS to the AMD SC520: A followup Report*, July 2005. http://www.linuxjournal.com/article/8310

[6] Advanced Micro Devices, Inc. *AMD Geode™ LX Processors Data Book*, June 2006. http://www.amd.com/files/connectivitysolutions/geode/geode_lx/33234E_LX_databook.pdf

[7] Advanced Micro Devices, Inc. *AMD Geode™ CS5536 Companion Device Data Book*, March 2006. http://www.amd.com/files/connectivitysolutions/geode/geode_lx/33238f_cs5536_ds.pdf

[8] Advanced Micro Devices, Inc. *AMD Geode™ GX and LX Processor Based Systems Virtualized PCI Configuration Space*, November 2006. http://www.amd.com/files/connectivitysolutions/geode/geode_gx/32663C_lx_gx_pciconfig.pdf

[9] LinuxBIOS, http://linuxbios.org. svn://openbios.org/repos/trunk/LinuxBIOSv2

[10] One Laptop Per Child. http://laptop.org

[11] OpenFirmware. http://firmworks.com svn://openbios.org/openfirmware

[12] Memtest86. http://memtest86.com

[13] Etherboot. http://www.etherboot.org/wiki/index.php

GANESHA, a multi-usage with large cache NFSv4 server

Philippe Deniel Thomas Leibovici Jacques-Charles Lafoucrière
CEA/DIF
{philippe.deniel,thomas.leibovici,jc.lafoucriere}@cea.fr

Abstract

GANESHA is a user-space NFSv2, NFSv3, and NFSv4 server. It runs on Linux, BSD variants, and POSIX-compliant UNIXes. It is available under the CeCILL license, which is a French transposition of the GPL and is fully GPL-compatible. The protocol implementation is fairly complete, including GSSAPI security hooks. GANESHA is currently in production at our site, where, thanks to a large cache and a lot of threads, it delivers up to a hundred thousand NFS operations per minute. This paper describes the current implementation as well as future developments. This includes GANESHA as a NFS Proxy server and NFSv4.1 enhancements, but also the access to LDAP and SNMP information using the file system paradigm.

1 Introduction

NFS is a well known and venerable network protocol which is used widely. NFSv4 is the latest version of the protocol. It fully reconsiders its semantic and the way NFS can be used.

We manage a huge compute center at CEA. In the past three years, we had to face a strong increase in the amount of data produced by our supercomputer, up to tens of terabytes a day. Archived results and files are stored in HPSS, a third-party vendor's HSM which had a NFS interface. NFS fits our need well in terms of files meta-data management, but there were several limitations in the product that made for a difficult bridge between the HSM and NFS, and we believed it was time to step to something new. The HPSS product has a user-space API, complete enough to do all manipulation on files and directories. The decision to write a brand new daemon to handle the NFS interface we needed to HPSS was natural, but the following ideas lead the design process:

- The new product should be able to manage very large data and meta-data caches (up to millions of records), to avoid congestion on the underlying file system.
- The new product should be able to provide the NFS interface we needed to HPSS, but should also be able to access other file systems.
- The new product should support the NFSv4 protocol, and its related features in term of scalability, adaptability, and security.
- The new product should be able to scale as much as possible: software congestion and bottlenecks should be avoided, the only limits would come from the hardware.
- The new product should be a free software program.
- The new product should be running on Linux, but portable to other Unix platforms.

These considerations drove the design of GANESHA. This paper will provide you with additional information about it. The generic architecture and the way it works will be described and you'll see how GANESHA can be turned into a "very generic" NFS server (using only POSIX calls from LibC) or a NFSv4 Proxy as well. Information will also be provided on the way to write packages to extend GANESHA in order to make it manage various names-spaces.

The paper first describes NFSv4 and the technical reasons that lead to a user-space NFS daemon. The architecture of the product is then detailed including the issues that were met and how they were solved. Some actual results are shown before concluding.

2 Why a NFSv4 server in User Space?

GANESHA is not a replacement for the NFSv4 server implemented in the kernel; it is a brand new program, with its advantages and disadvantages. For some aspects, the NFSv4 server in the kernel should be more efficient, but there are several domains (for example building a NFSv4 Proxy server) in which the user-space approach will provide many interesting things.

First of all, working in user space makes it possible to allocate very large piece of memory. This memory can then be used to build internal caches. Feedback of using GANESHA in production showed that 4 Gigabytes were enough for making a million-entry cache. On a x86_64 platform, it is possible to allocate even bigger memory chunks (up to 16 or 32 GB, depending on the machine's resources). Caching about 10 million entries becomes possible.

A second point is portability. If you write kernel code, then it will be acquainted with the kernel's structure and it won't be possible to port it to a different OS. We kept Linux (i686 or x86_64) as the primary target, but we also wanted to compile and run it on different architectures, keeping them as secondary targets. Most of the Free Software Community is very close to Linux, but there are other free operating systems (FreeBSD or OpenSolaris) and we have wanted to be compatible with them since the beginning of the project. Another consideration is the code itself: something that compiles and runs on different platforms is generally safer than a "one target" product. Our experience as developers showed that this approach always pays back; it often reveals bugs that would not have been so easily detected on Linux, because resources are managed differently. Portability doesn't only mean "running on several OSes," for a NFS server it also means "managing different file systems." The NFSv4 semantics bring new ideas that need to be considered there. The NFSv2 and NFSv3 protocols have semantics very close to the way Unixes manage file systems. Because of this, it was almost impossible to have NFS support for a non UNIX-related file system. One design consideration of NFSv4 was to make the protocol able to manage as many file systems as possible. Because of this, it requires a very reduced subset of file/directory attributes to be supported by the underlying file system and can manage things as simple as a FAT16 file system (which has almost none of the attributes you expect in "regular" file systems). When designing GANESHA, we wanted to keep this idea: managing as many file systems as possible. In fact, it is possible with the NFSv4 semantics to manage every set of data whose organization is similar to a file system: trees whose nodes are directories and leaves are files or symbolic links. This structure (that will be referenced as the *name-space* structure in this paper) maps to many things: files systems of course, but also information accessible through a SNMP MIB or LDAP-organized data. We choose to integrate this functionality to GANESHA: making it a generic NFSv4 server that can manage everything that can be managed by NFSv4. Doing this is not very easy within the kernel (kernel programming is subject to lots of constraints): designing the daemon for running in user space became then natural.

A last point is also to be considered: accessing services located in user space is very easy when you already are in user space. NFSv4 support in the kernel introduced the *rpc_pipefs* mechanism which is a bridge used by kernel services to address user-space services. It is very useful for managing security with kerberos5 or when the *idmapd* daemon is asked for a user-name conversion. This is not required with GANESHA: it uses the regular API for the related service.

These reasons naturally lead the project to a user-space daemon. We also wanted to write something new and open. There was already an efficient support of NFSv4 support within kernel code. Rewriting something else would have had no sense. This is why GANESHA is a user-space daemon.

3 A few words about NFSv4

NFS in general, and more specifically NFSv4, is a central aspect to this paper. People are often familiar with NFS, but less are aware of the features of NFSv4.

NFSv2 was developed by Sun Microsystems in 1984. It showed limits and this lead to the birth of NFSv3, which was designed in a more public forum by several companies. Things were a bit different with NFSv4. The protocol has been fully developed by an IETF working group (IETF is responsible for standardization of protocol like IPv4, IPv6, UDP, TCP, or "higher-level" things like FTP, DNS, and HTTP). The design began with a birds-of-a-feather meeting at IETF meetings. One of the results was the formation of the *NFS version 4* working group in July, 1997.

Goals of the working group when designing the protocol were:

- improve access and performance on the Internet;
- strong security with negotiation built into the protocol;
- easier cross-platform interoperability;
- the protocol should be ready for protocol extensions.

NFSv4 integrates features that allow it to work correctly on a WAN, which is a network with low bandwidth and high latency. This is done through using experience obtained with protocols like WebNFS. NFSv4 will then use compound requests to limit messages and send as much information as possible in each of them. To reduce traffic, the caching capability were truly extended, making the protocol ready for implementation of very aggressive caching and an NFSv4 proxy server.

Scalability and availability were improved, too; a strong stateful mechanism is integrated in the protocol. This is a major evolution compared to NFSv2 and NFSv3, which were stateless protocols. A complex negotiation process occurs between clients and server. Due to this, NFSv4 can allow a server with a strong load to relocate some of its clients to a less busy server. This mechanism is also used when a client or server crash occurs to reduce the time to full recovery on both sides.

Security is enhanced by making the protocol a connection-oriented protocol. The use of RPCSEC_GSS is mandatory (this protocol is an evolution of ONC/RPC that supports extended security management—for example the krb5 or SPKM-3 paradigm—by use of the GSSAPI framework) and provides "RPC-based" security. The protocol is connection-oriented, and will require TCP (and not UDP like NFSv2 and NFSv3), which makes it easier to have connection-based security.

The structure and semantics of NFSv3 were very close to those of UNIX. For other platforms, it was difficult to "fit" in this model. NFSv4 manages attributes as bitmaps, with absolutely no link to previously defined structures. Users and groups are identified as strings which allow platforms that do not manage uid/gid like UNIX to interoperate via NFSv4.

The protocol can be extended by support of "minor versions." NFSv4 is released and defined by RFC3530, but evolutions are to be integrated in it, providing new features. For example, the support of RDMA, the support of the PNFS paradigm, and the new mechanism for "directory delegation" are to be integrated in NFSv4. They will be part of NFSv4.1, whose definition is in process.

4 Overview

This section describes the design consideration for GANESHA. The next sections will show you how these goals were achieved.

4.1 The CeCILL License

GANESHA is available as a Free Software product under the terms of the CeCILL license. This license is a French transposition of GPL made by several French research organizations, including CEA, CNRS, and INRIA. It is fully GPL-compatible.

The use of the GNU General Public License raised some legal issues. These issues lead to uncertainties that may prevent contributions to Free Software. To provide better legal safety while keeping the spirit of these licenses, three French public research organizations, the CEA, the CNRS, and INRIA, have launched a project to write Free Software licenses conforming to French law. CEA, CNRS, and INRIA released CeCILL in July, 2004. CeCILL is the first license defining the principles of use and dissemination of Free Software in conformance with French law, following the principles of the GNU GPL. This license is meant to be used by companies, research institutions, and all organizations willing to release software under a GPL-like license while ensuring a standard level of legal safety. CeCILL is also perfectly suited to international projects.

4.2 A project on Open-Source products

GANESHA was fully written and developed using Free Software. The resources available for system programming are huge and comprehensive, and this made the task much easier on Linux than on other Unixes.

The tools used were:

- *gcc* (of course...)

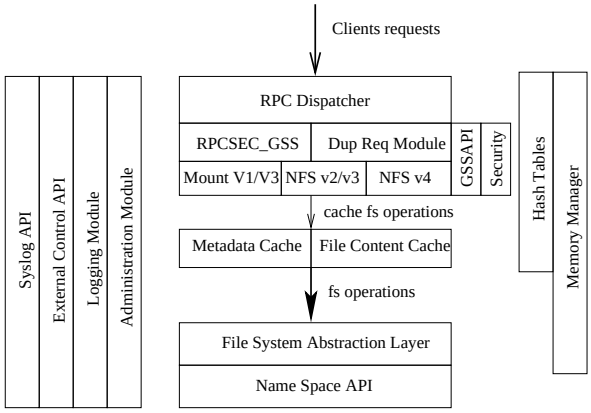


Figure 1: GANESHA's layered architecture

- *gdb* for debugging, often used jointly with Electric Fence or the Dmalloc library for memory debugging.
- *valgrind* for caring about memory leaks.
- *doxygen* for generating the various documents about the APIs' calls and structures.
- *GIT* as source code repository manager.
- *PERL* and *SWIG* to wrap API calls in order to write non-regression scripts.
- *Connectathon test suite* which is a test suite designed for the validation of NFS client-server behavior.
- *PyNFS* a non-regression test tool written in Python by the CITI folks.[1]

4.3 A layered product

GANESHA is designed as a layered product. Each layer is a module dedicated to a specific task. Data and meta-data caching, RPCSEC_GSS and protocol management, accessibility to the file system... All these functionalities are handled by specific modules. Each module has a well defined interface that was designed before starting to write a single line of code. Such a modular design is good for future code maintenance. Furthermore, one can write new algorithms within a layer without changing the rest of the code. A better description is that cache management could change the cache layers, or a different name-space could be managed, but these changes should not impact the other modules. Efforts were made to reduce adherences between layers. This was costly at the beginning of the project, but on a mid-range time scale, it appeared that this simplified a lot in the rest of the project. Each layer could be developed independently, by different developers, with their own validation and non-regression tests. A "global make" step can then re-assemble all the pieces. It should be reduced if all of them complete their validation tests.

A few modules are the very core of GANESHA:

- The Buddy Malloc module manages the memory used by GANESHA.
- The RPCSEC_GSS module handles the data transport via the RPCSEC_GSS protocol. It manages security by accessing the security service (usually krb5, SPKM-3, or LIPKEY).
- The NFS protocol modules perform the management of the structures used for the NFS messages.
- The Cache Inode Layer manages a very large cache for meta-data.
- The File Content Layer manages data caching. It is closely acquainted with the Cache Inode Layer.
- The File System Abstraction Layer is a very important module: it wraps, via a well defined interface, the calls to access a name-space. The objects it addresses are then cached by the Cache Inode and File Content layers.
- The Hash Table Module provides Red-Black-Trees-based hash tables. This generic module is widely used within GANESHA to provide associative addressing.

These modules will be discussed in more details in the next sections.

4.4 Managing memory

The main issue is memory management. Almost all modules within GANESHA's architecture will have to perform dynamic memory allocation. For example, a thread managing a NFS request may need to allocate a buffer for storing the requested result. If the regular

[1] CITI's site contains bunches of interesting stuff for people interested in NFSv4.

LibC malloc/free calls are used, there are risks of fragmenting memory because some modules will allocate large buffers when others will use much smaller ones. This could lead to a situation where part of the memory used by the program is swapped to disk, and performance would quickly drop.

For this reason, GANESHA implements its own memory manager. This module, which is used by all the other parts of GANESHA, allows each thread to allocate its own piece of memory at startup. When a thread needs a buffer, it will look into this space to find an available chunk of the correct size. This allocation is managed by the Buddy Malloc algorithm, the same that is used by the kernel. Use of the syscall *madvise* is also made to tell the Linux memory manager not to move the related pages. The behavior of the daemon towards memory will then be to allocate a single large piece of memory. If there is no other "resource consuming" daemon running on the same machine, the probability for this piece of memory not to be swapped is high. This will maintain performance at a good level.

4.5 Managing the CPU resource

The second resource is the CPU. This is much easier to manage than memory. GANESHA is massively multi-threaded, and will have dozens of threads at the same time (most of them are "worker threads," as we'll see later). POSIX calls for managing threads help us a lot here, we can use them to tell the Linux scheduler not to manage the pack of threads as a whole, but to consider each of them separately.[2] With a multi-processor machine, such an approach will allow the workload to "spread across" all of the CPUs. What is also to be considered is potential deadlocks. In a multi-threaded environment, it is logical to have mutexes to protect some resources from concurrent accesses. But having bunches of threads is not useful if most of them are stuck on a bottleneck. Design considerations were taken into account to avoid this situation.

First, reader/writer locks were preferred to simple mutexes. Because the behavior of reader/writer locks may differ from one system to another, a small library was written to provide this service (which was a required enhancement in terms of portability).

Second, if threads share resources, this common pool could turn to a bottleneck when many threads exist together. This was avoided by allocating resources per thread. This consideration has a strong impact on the threads' behavior, because there can't be a dedicated garbage collector. Each thread has to perform its own garbage collection and has to reassemble its resources regularly. To avoid congestion, some mechanism (located on the "dispatcher thread" described below) will prevent too many threads from performing this operation at the same time (a period during which they are not available for doing their "regular" job). Cache layers that require this kind of garbage collection to be done have been designed so that this process could be divided in several steps, each undertaken by a separate agent. Experience "in real life" shows that this solution was suitable when the number of threads is large compared to the number of threads allowed to start garbage collecting (60 threads running concurrently when 3 could stop working at the same time). This experience shows that the required memory chunk was much less that what is needed for a single request (about 20 times the size). In this situation, the impact of memory management is almost invisible: an incoming request finds a non-busy thread most of the time. Side effects will only become visible under a very large load (hundreds to thousands of requests per second).

4.6 The Hash Tables: a core module for associative addressing

Associative addressing is a service that is required by many modules in GANESHA—for example, finding an inode knowing its parent and name, or finding the structure related to a NFSv4 client, knowing its client ID. The API for this kind of service is to be called very often: it has to be very efficient to enhance the daemon's global performance. The choice was made to use an array of Red-Black Trees.[3] RBTs have an interesting feature: they re-balance themselves automatically after add/update operations and so stay well balanced. RBTs use a computed value, defined as the *RBT value* in this document, to identify a specify contiguous region of the tree. Several entries stored in the RBT can produce the same RBT value, they'll reside the same area, but this will decrease the performance. Having a function to compute "well diversified" RBT values is then critical.

[2] This is the `PTHREAD_SCOPE_SYSTEM` behavior which is used here, as opposed to the `PTHREAD_SCOPE_PROCESS` policy that would not lead to the expected result.

[3] We'll use the abbreviation RBT for Red-Black Tree in the rest of this paper.

This supposes an actual knowledge of the data on which the value is computed. Because of this it is hard to have a "generic RBT value function," a new one is to be developed for each use.

Bottlenecks could occur if a single RBT is used: several threads could perform add/update operations at the same time, causing a conflicting re-balance simultaneously. It then appears that RBTs are to be protected by read/writer locks and this could quickly become a bottleneck. Working around this issue is not difficult: using several RBTs (stored in an array) will solve it. If the number of RBTs used is large (more than 15 times bigger) that the number of concurrent threads that can access them, the the probability of having two of them working on the same tree becomes pretty small. This will not use more memory: each of the 15 (or more) trees will be 15 times smaller than the single one would have been. There is an inconvenience: an additional function is required to compute the index for the RBT to be used. Implementing two functions is then needed for a single hash table: one for computing the index, the other to compute the RBT value. They must be different enough to split data across all the trees. If not, some RBTs would be very small, and others very large. Experience shows that specific non-regression tests were necessary to check for the "independence" of these two functions.

4.7 A massively multi-threaded daemon

GANESHA is running lots of threads internally. As shown in the previous sections, most of its design consideration were oriented to this massively multi-threaded architecture. The threads are of different types:

- GANESHA supports NFSv2, NFSv3, NFSv4, and the ancillary protocol MOUNT PROTOCOL v1 and v3. The *dispatcher* thread will listen for incoming NFS/MOUNT requests, but won't decode them. It will choose the least busy worker and add the request to its lists of requests to process. Duplicate request management is done here: this thread keeps track of the previously managed requests by keeping the replies sent within the last 10 minutes (they are stored in a hash table and addressed with the RPC Xid[4] value). Before associating a worker with a request, it looks at this DRC.[5] If a matching RPC Xid is found, then the former reply is sent back again to the client. This thread will use the RPCSEC_GSS layer, mostly.

- The *worker* threads do most of the job. Many instances (several dozen) of this kind of thread exist concurrently. They wait for the dispatcher thread to provide them with a request to manage. They will decode it and use Cache Inode API and File Content API calls to perform the operation required for this request. These threads are the very core of the NFS processing in GANESHA.

- The *statistics manager* collects stats from every layer for every thread. It periodically writes down the data in CSV format[6] for further treatment. A dedicated PERL script, `ganestat.pl`, is available with the GANESHA rpm as a "pretty printer" for this CSV file.

- The *admin gateway* manages a dedicated protocol. This allows administrative operations to be done remotely on the daemon. These operations include flushing caches, syncing data to FSAL storage, or performing a slow and clean shutdown. The `ganeshadmin` program, provided with the distribution, is used to interact with this thread.

4.8 Dealing with huge caches

As stated above, GANESHA uses a large piece of memory to build large caches. Data and meta-data caches will be the largest caches in GANESHA.

Let's focus first on the meta-data cache, located in the Cache Inode Layer. Each of its entries is associated with an entry in the name-space (a file, a symbolic link, or a directory[7]). This entry is itself associated with a related object in the File System Abstraction Layer (see next section) identified by a unique FSAL handle. The meta-data cache layer will map in memory the structure it reads from the FSAL calls, and it tries to keep in memory as many entries as possible, with their parent-children dependencies. Meta-data cache use hash tables

[4] See the definition of ONC/RPC protocol for details on this.

[5] Duplicate Request Cache.

[6] Comma Separated Value, an ASCII based format for storing spreadsheets.

[7] For the current version, objects of type socket, character, or device are not managed by GANESHA.

intensively to address the entries, using the FSAL handle to address the entry associatively. With the current version of GANESHA, a simple write-through cache policy is implemented. The attributes kept for each object (the file attributes and the content of the directories) will expire after a configurable grace period. If expired, they'll be renewed if they are accessed before being erased from the cache. Garbage collection is more sophisticated. Because there is no common resources pool, each thread has to perform garbage collection itself. Each thread will keep a LRU list of the entries on which it works. A cached entry can exist only within one and only one of these lists, so if a thread accesses an entry which was previously accessed by another, it acquires this entry, forcing the other thread to release it. When garbage collection starts, the thread will go through this list, starting from the oldest entry. It then use a specific garbage policy to decide whether each entry should be kept or purged. This policy is somewhat specific. The meta-data cache is supposed to be very large (up to millions of entries) and no garbage collection will occur before at least 90% of this space is used. We choose to keep as much as possible the "tree topology" of the name-space viewed by the FSAL in the cache. In this topology, nodes are directories, and leaves are files and symbolic links. Leaves are garbage collected before nodes. Nodes are garbage only when they contain no more leaves (typically an empty directory or a directory where all entries were previously garbaged). This approach explicitly considers that directories are more important than files or symbolic links, but this should not be an issue. Usually, a name-space will contain a lot more files than directories, so it makes sense to garbage files first: they occupy most of the available space. Because the cache is very large, parts of it tend to be "sleeping areas" that are no longer accessed. The garbage collection routine within each worker thread, which manages the oldest entries first, will quickly locate these and clean them. With our workload and file system usage, this policy revealed no problem. When the garbage collection's high water mark is reached, the number of entries cached begins to oscillate regularly between low water mark and high water mark. The period of the oscillation is strongly dependent on the average load on the server.

The data cache is not managed separately: if the content of a file is stored in data cache, this will become a characteristic of the meta-data cached entry. The data cache is then a 'child cache' to the meta-data cache: if a file is data-cached, then it is also meta-data cached. This avoid incoherencies between this two caches since they are two sides of the same coin. Contents of the files which are cached are stored in dedicated directories in a local file system. A data-cache entry will correspond to two files in this directory: the index file and the data file. The index files contain the basic meta-data information about the file; the most important one is its FSAL handle. The data file is the actual data corresponding to the cached file. The index file is used to rebuild the data-cache, in the event that the server crashes without cleanly flushing it: the FSAL Handle will be read from this file and then the corresponding meta-data cache entry will be re-inserted as well, making it point to the data file for reconstructing the data cached entry. Garbage collection is performed at the same time as meta-data cache garbage collection. Before garbaging files, the meta-data cache asks the data cache if it knows this entry or not. If not, regular meta-data garbage collection is performed. If yes, the meta-data cache asks the data cache to apply its garbage policy on it, and eventually flush or purge it. If the file is cleaned from the data cache, it can be garbaged from meta-data cache. A consequence of this is that a file which has an active entry in the data cache will never be cleaned from the meta-data cache. This way of working fits well with the architecture of GANESHA: the worker threads can manage the data cache and meta-data cache at the same time, in a single pass. As stated above, the two caches are in fact the same, so no incoherence can occur between them. The data cache has no scalability issue (the paths to the related files are always known by the caches) and does not impact the performance of the meta-data cache. The policy used for data cache is "write-back" policy, and only "small" files (smaller than 10 MB) will be managed; others would be accessed directly, ignoring the data cache. Smarter or more sophisticated algorithms can be implemented—for example, the capability, for very large files, to cache a region of the file but not the whole file. This implementation could be linked to NFSv4 improvements like NFSv4 named attributes or the use of the PNFS paradigm (which is part of the NFSv4.1 draft protocol).

5 File System Abstraction Layer

FSALs (or File System Abstraction Layers) are a very important module in GANESHA. They exist in different incarnations: HPSS FSAL, POSIX FSAL, NFSv4

Proxy FSAL, SNMP FSAL, and LDAP FSAL. They provide access to the underlying file name-space. They wrap all the calls used for accessing it into a well defined API. This API is then used by the Cache Inode and File Content module. FSAL can use dedicated APIs to access the name-space (for example, the SNMP API in the case of SNMP FSAL), but this API will completely hidden from the other modules. FSAL semantics are very close to the NFSv4 semantics, an approach that is repeated in the Cache Layers. This uniformity of semantics, close to native NFSv4, makes the implementation of this protocol much easier. Objects within FSAL are addressed by an FSAL Handle. This handle is supposed to be persistent-associated with a single FSAL object by an injective relationship: two different objects will always have different handles. If an object is destroyed, its handle will never be re-used for another FSAL object. Building a new FSAL is the way to make GANESHA support a new name-space. If the produced FSAL fits correctly with the provided non-regression and validation tests, then the GANESHA daemon need only be recompiled with this new FSAL to provide export over NFS for it. Some implementation documents are available in the GANESHA distribution. External contributors may actively participate to GANESHA by writing additional FSALs. Templates for FSAL source code are available in the GANESHA package.

5.1 The HPSS FSAL

This FSAL is not related to Free Software, but a few words must be said for historical reasons, because it strongly contributed to the origin of the project. We are using the HSM named HPSS,[8] a third-party vendor product sold by the IBM company. This HSM manages a name-space, accessible in user space via dedicated API, which fully complies with the FSAL pre-requisites. The name-space is relatively slow, and this led us to improve the caching features in GANESHA. This module is available, but not within the regular distribution of GANESHA (you need to have HPSS installed to compile it with the HPSS API libraries).

5.2 The POSIX-based FSAL

This flavor of FSAL uses the regular POSIX calls (*open*, *close*, *unlink*, *stat*) from LibC to manage file system objects. All the file systems managed by the machine on which the daemon is running (depending on its kernel) will be accessible via these functions; using them in GANESHA provides generic NFS access to all of them. The inconvenience is that POSIX calls often use the pathnames to the objects to identify them. This is no persistent information about the object (a rename could be performed on it, changing its name). This does not fit with the pre-requisite to build FSAL, as described in the previous subsection. Another "more persistent" identifier is to be found. The choice was made to use an ancillary database (basically a PostgreSQL base) to build and keep the identifier we need. The tuple (inode number, file system ID, ctime attributes) is enough to fully identify an object, but the name should be used to call the POSIX functions. The database will keep parenthood relationship between objects, making it possible to rebuild the full path to it, by making a kind of "reverse lookup" when needed. SQL optimization and pathname caching were used a lot in the module. A complete description of the process would require a full paper. Why develop such a module when it could be much easier to use the NFS interface in the kernel? The answer is linked with the resource we use at our compute center.

GANESHA can access more file systems than most available kernels at our site. We had the need to access the LUSTRE file system, but some machines were not LUSTRE clients. In most cases, they are not Linux machines. We strongly needed them to be able to access the LUSTRE name-space. This could not be done via NFS kernel support: this NFS implementation uses the VFS layer a lot, a part of the kernel that is often bypassed by the LUSTRE implementation for optimization. This approach, using the simple POSIX calls to access LUSTRE from GANESHA, was quick to write and not very costly.

This module is available.

5.3 The NFSv4 Proxy FSAL

When designing GANESHA, we had one thought: having a NFSv4 proxy would be great. NFSv4 has lots of features that are designed for implementing aggressive cache policy (file delegation is a good example of this feature). GANESHA is designed to manage huge caches. The "wedding" seems very natural here. The NFsv4 Proxy FSAL wraps NFSv4 client calls to FSAL calls. It turns the back-end part of GANESHA into a

[8] HPSS stands for *High Performance Storage System*.

NFSv4 client, turning the whole daemon into a NFSv4 proxy server. The mechanism of file delegation is a feature in NFSv4 that is quite interesting here. It allows a file to be "fully acquired" by a client for a given period of time. Operations on files, such as IO operations and modification of its attributes, will be done on the client directly, without disturbing the server; that guarantees that no other clients will access it. Depending on the kind of delegation used, the server may use transient callbacks to update information about the file. When the delegation ends, the server recovers the file, getting the new state for the file from the client. Delegation, used jointly with GANESHA meta-data and data caches, is very efficient: accessing a file's content will be done though data cache, once a delegation on the file has been acquired. The policy for the NFSv4 Proxy FSAL will be to acquire as many delegations as possible, populating the GANESHA's caches. With a well populated cache, GANESHA will become able to answer by proxy many requests. In NFSv4.1, a new feature is added: the directory delegation. This will allow the content of directories to be delegated and acquired by clients in the same way that file contents are. Used with GANESHA's meta-data cache, this feature will be very interesting.

This module is still under development.

5.4 The "Ghost FS" FSAL

This FSAL is a very simple one and is not designed for production use. It just emulates the behavior of a file system in memory, with no persistent storage. The calls to this FSAL are very quick to return because all the work is done in memory, no other resources are used. Other FSALs are always much slower than the cache layer.[9] It is hard to evaluate meta-data and cache modules performances. With the "Ghost FS" FSAL, calls to these layers can be easily qualified, and it is possible to identify the most costly calls, and thus to optimize GANESHA.

This module is available.

5.5 The LUSTRE FSAL

As mentioned above, LUSTRE is a file system we use a lot, and we would like to access it from machines that are not LUSTRE clients. We already developed the POSIX FSAL for this, but having something more acquainted with LUSTRE would be nicer. Having a user-space LUSTRE API able to perform operations in a handle-based way would be something very interesting: it would allow us to wrap the API to a LUSTRE FSAL, making the access to this file system via the GANESHA NFSv4 interface much more efficient than the one we have with the POSIX FSAL. We also hope to use the NFSv4 named attributes[10] to provide clients for LUSTRE-specific information about the file (the resident OST[11] of the file is a good example).

This module is under definition. It will be finalized as soon as a handle-based LUSTRE API is available.

5.6 The SNMP FSAL

The SNMP protocol organizes sets of data as trees. The overall structure of the trees is defined by files named MIB.[12] Knowing the MIB yields the ability to compute the OID[13] to access a given management value. This OID is basically a list of numbers: each of them identifies a node at the given level in the tree, and the last one identifies the leaf where the data resides. For example, `.1.3.6.1.4.1.9362.1.1.0` identifies the Uptime value in the SNMPv2 MIB. This OID is used to query a SNMP agent about the time since the last reboot of the machine. OIDs can also be printed in a "symbolic" way, making them more human readable. In the previous example, `.1.3.6.1.4.1.9362.1.1.0` is printed as *SNMPv2-MIB::system.sysUpTime*. This tree structure is in fact a name space: each SNMP-accessible variable can be seen as a "file object" whose content is the value of the variable. There are "directories" which are the nodes in the MIB structure. OIDs are very good candidates for being handles to SNMP objects, and are to be mapped to names (the symbolic version of the OID). This clearly shows that SNMP has enough features to build an FSAL on top of it. Using it with GANESHA will map the SNMP information into an NFS export, able to be browsed like a file system. It is then possible to browse SNMP in a similar way to the `/proc` file system. In our example, Handle `.1.3.6.1.4.1.9362.1.1.0` would

[9] Otherwise there would have been no need for caches...

[10] which are basically the way NFSv4 manages extended attributes.

[11] Object Storage Target: the way LUSTRE views a storage resource.

[12] Management Information Base.

[13] Object ID.

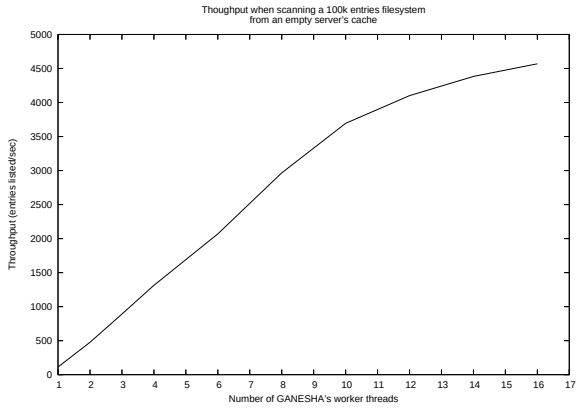


Figure 2: Performance with an empty metadata-cache

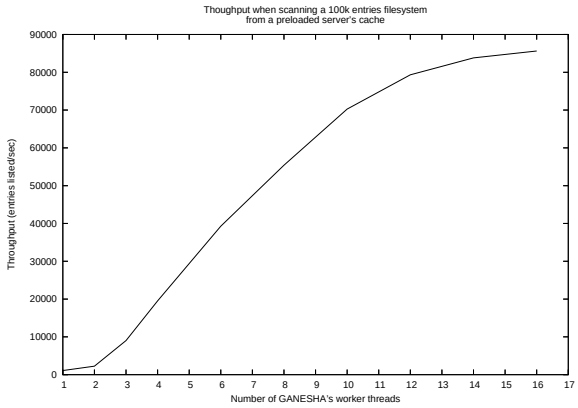


Figure 3: Performance with a preloaded metadata-cache

map to `(mounted NFS PATH)/SNMPv2-MIB/system/sysUpTime`. A read operation on `SNMPv2-MIB/system/sysUpTime` would yield the corresponding value.

Some SNMP values are settable: in this approach, they could be changed by writing to the file corresponding to them.

This module is under development.

5.7 The LDAP FSAL

The idea for this FSAL is the same as for the SNMP FSAL. LDAP has a name-space structure and is accessible via a user-space API. This FSAL simply wraps this API to provide FSAL support, then NFS support via GANESHA for LDAP. LDAP information will then be browsed like `/proc`, via NFS.

This module is under development.

6 Performances and results

In this section, we will show GANESHA's scalability feature by an actual test. The test is as follows: a specific tool was written to perform, in a multi-threaded way (the number of threads is configurable) what `find . -ls` does, which is scanning a whole large tree in a name-space. This tree contained 2220 directories on 3 levels; each of them contained 50 files (which means more that 110,000 files were in the whole tree). The test utility ran on several client nodes (up to 4 machines) using the same server. The multi-threaded test utility was run of each of these 4 clients with 64 threads each. This was equivalent to 256 cache-less clients operating concurrently. The server machine was a IBM x366 server with four Intel Xeon 3 GHz processors and 4 GB of RAM, running GANESHA built with the POSIX FSAL. Two groups of measurements were made. The first one is done with a server whose meta-data cache is empty (Figure 2), and the second (Figure 3) with the same server with a preloaded cache. In this second step, the read entries exist in the memory of the server, and the performance of the meta-data cache can be compared to the raw FSAL performances.

Figure 2 shows that saturation of the FSAL occurs quickly. Increasing the number of worker threads increases the performance, but no larger throughput than 5,000 entries read per second can be reached. Observations made on the server showed that no CPU or memory contention led to this saturation effect. The reason was that the POSIX FSAL on top of the underlying POSIX calls did not scale to these values.

Figure 3 shows different results. Due to the meta-data cache, most of the operations are done directly in memory, reducing greatly the calls to POSIX FSAL. The throughput raises up to 90,000 entries read per second. The dependence between this throughput and the number of worker threads is linear, which shows the scalability of the process. After 11 worker threads, we can't see such linearity. The reason for this was due to CPU congestion. The OS could not allocate enough CPU time to all the workers, and they start waiting to be scheduled. This test should be performed on a larger platform.

This test shows that the multi-thread architecture in

GANESHA provides good scalability.

7 Conclusion and perspectives

GANESHA has been in production at our site for more than one full year. It fits the needs we had when the decision was taken to start the project. Its large cache management capability allowed an increase of the incoming NFS requests on the related machines, a need that was critical for several other projects.

When the product started in full production, in January, 2006, this provided us with very useful feedback that helped in fixing bugs and improved the whole daemon. Thanks to this, GANESHA is a very stable product in our production context at our site. Making GANESHA Free Software is an experience that will certainly be very positive; we expect the same kind of feedback from the Open Software community. GANESHA can also be of some interest for this community; we actually believe that is could serve well as a NFSv4 Proxy or as an SNMP or LDAP gateway.

NFSv4 is also a very exciting protocol, with plenty of interesting features. It can be used in various domains and will probably be even more widely used that the former version of NFS. Lots of work is done around this protocol, like discussion about implementing its features or extending it with new features (see NFSv4.1 drafts). GANESHA will evolve as NFSv4 will. We hope that you will find this as exciting as we did, and we are happy to share GANESHA with the community. We are eagerly awaiting contributions from external developers.

References

[1] S. Shepler, B. Callaghan, D. Robinson, Sun Microsystems Inc., C. Beame, Hummingbird Ltd., M. Eisler, D. Noveck, Network Aplliance Inc. *"Network File System (NFS) version 4 Protocol," RFC 3530*, The Internet Society, 2003.

[2] Callaghan, B., Pawlowski, B. and P.Staubach, *"NFS Version 3 Protocol Specification," RFC 1813*, The Internet Society, June, 1995.

[3] Sun Microsystems, Inc., *"NFS: Network File System Protocol Specification," RFC 1094*, The Internet Society, March, 1989.

[4] Shepler, S., *"NFS Version 4 Design Considerations," RFC 2624*, The Internet Society, June, 1999.

[5] Adams, C., *"The Simple Public-Key GSS-API Mechanism (SPKM)," RFC 2025*, The Internet Society, October, 1996.

[6] Eisler, M., Chiu, A. and L. Ling, *"RPCSEC_GSS Protocol Specification," RFC 2203*, The Internet Society, September, 1997.

[7] Eisler, M., *"NFS Version 2 and Version 3 Security Issues and the NFS Protocol's Use of RPCSEC_GSS and Kerberos V5," RFC 2623*, The Internet Society, June, 1999.

[8] Linn, J., *"Generic Security Service Application Program Interface, Version 2, Update 1," RFC 2743*, The Internet Society, January, 2000.

[9] Eisler, M., *"LIPKEY—A Low Infrastructure Public Key Mechanism Using SPKM," RFC 2847*, The Internet Society, June, 2000.

[10] Shepler, S., Callaghan, B., Robinson, D.,Thurlow, R., Beame, C., Eisler, M. and D. Noveck, *"NFS version 4 Protocol," RFC 3010*, The Internet Society, December, 2000.

[11] B. Callaghan, *"NFS Illustrated,"* Addison-Wesley Longman Ltd., Essex, UK, 2000.

[12] CITI. *Projects: NFS Version 4 Open Source Reference Implementation*, `http://www.citi.umich.edu/projects/nfsv4/linux`, June, 2006.

[13] Connectathon. *Connectathon web site*, `http://www.connectathon.org`.

[14] S. Khan. *"NFSv4.1: Directory Delegations and Notifications," Internet draft*, `http://tools.ietf.org/html/draft-ietf-nfsv4-directory-delegation-01`, Mar 2005.

[15] Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, *"The NFS Version 4 Protocol,"* In Proceedings of Second International System Administration and Networking (SANE) Conference, May, 2000.

[16] A. Charbon, B. Harrington, B. Fields, T. Myklebust, S. Jayaraman, J. Needle, *"NFSv4 Test Project,"* In Proceedings to the Linux Symposium 2006, July, 2006.

[17] P. Åstrand, *"Design and Implementation of a Test Suite for NFSv4 Servers,"* September, 2002.

[18] CEA, CNRS, INRIA. *"CeCILL and Free Software,"* `http://www.cecill.info/index.en.html`.

Why Virtualization Fragmentation Sucks

Justin M. Forbes
rPath, Inc.
jmforbes@rpath.com

Abstract

Mass adoption of virtualization is upon us. A plethora of virtualization vendors have entered the market. Each has a slightly different set of features, disk formats, configuration files, and guest kernel drivers. As concepts such as Virtual Appliances become mainstream, software vendors are faced with new challenges. Previously, software vendors had to port their application to multiple operating systems: Solaris, Linux, AIX, etc. The new "port" becomes one where software vendors will be expected to produce images that drop-in to VMware, Xen, Parallels, SLES, RHEL, and even Microsoft Virtual Server.

This paper will explore the state of existing virtualization technology in meeting the goal of providing ready-to-run guest images. This includes: comparing, contrasting, and poking fun at virtual disk formats; bemoaning the assortment of kernel drivers needed to improve performance in a guest (vmware-tools, paravirt drivers...); dark muttering about incompatibilities between Xen guests and hosts; and lamenting all the different configuration files that define a guest.

Virtualization has moved into the mainstream of computing. Most businesses are no longer asking if they should deploy a virtualization solution; they are instead asking which vendors support the technologies they have already implemented. In many ways, this is a great new age for software vendors. With virtualization technology so common, it makes it possible to reduce the costs associated with supporting a product on a large assortment of operating systems. Software vendors can now bundle just the right amount of operating system required to support their software application in a software appliance model. Distributing a software appliance allows vendors to fully certify one stack, without the worry about which packages or versions a particular operating system distribution chooses to ship. The extensive QA and support models that go along with shipping a separate application are drastically simplified. Software vendors no longer need to decide which operating systems to support, the new question is "which virtualization technologies do I support?"

This question should be easy to answer. Unfortunately, it is becoming increasingly difficult. It does not have to be. The hypervisor is the new platform, and many vendors have entered the market, with more vendors on the way. Each vendor offers products to meet a similar requirement: Allow fully isolated containers, or virtual machines, to consume the resources they require, without stepping on other containers.

The advantages of virtualization are many. To the software consumer, virtualization takes away much of the concern surrounding the full stack. The fact that different applications may require conflicting library support is no longer a concern. The ability to better manage resources, increasing utilization of hardware without increasing risk of multiple applications stepping on each other is a tremendous benefit. Live migration, the ability to move virtual machines across physical hosts in real time, substantially increases availability. It is possible to maintain both performance and availability with a fraction of the hardware resources that were once required.

In the software appliance model, vendor relationships are improved as the customer can go to a single vendor for support, without playing intermediary between the application vendor, tools vendors, and operating system vendors. Software consumers need not worry about testing and patching large numbers of security and bug fixes for software which gets installed with most general purpose operating systems, but is never used or installed within their virtualized environment.

To the software producer and distributor, virtualization means simplified development, QA, and testing cycles. When the application ships with its full stack, all of the time required to ensure compatibility with any number

of supported platforms goes away. The little compromises required to make sure that applications run reliably across all platforms have a tendency to ensure that those applications do not run optimally on any platform, or increase the code complexity exponentially. This pain goes away when the platform is a part of the application, and unnecessary components which exist in a general purpose application are no longer present. Software vendors can distribute fully tested and integrated appliances, and know exactly what components are on the appliance without concern that a critical component was updated to fix a bug in some other application unrelated to what the vendor's application provides or supports.

With so many benefits to virtualization, and so many options available to the consumer, what could possibly go wrong? No one ever likes the answers to that question. The proliferation of options in the virtualization market has brought a new kind of insanity for software vendors. "Which technologies do I provide ready-to-run guest images for?" The simple answer should be all of the major players. Providing choice to customers with minimal effort is a great thing. Unfortunately, the effort is not so minimal at the moment.

Each vendor has a business need to distinguish itself by providing unique features or a unique combination of features. Unfortunately for the consumer, even generic features are provided in unique ways by each virtualization provider, needlessly complicating life both for the software vendor and the end user.

This doesn't have to be so hard.

1 Disk Formats

There are several possibilities for the virtual machine disk format, none of which is universally supported. While most of the commonly-used formats offer a similar set of features, including sparse allocation and copy on write or some form of snapshot, they are not directly interchangeable. VMware's VMDK and Microsoft's VHD are among the most common formats supported. The QCOW format also offers similar functionality, though it should be noted that there are now multiple incompatible versions of the QCOW formats, making QCOW more of a format family than a format. The disk format will typically include the raw file systems or hard disk image, a bit of metadata describing the supported features, versioning, and creation method, as well as specific implementation metadata in the case of sparse allocation or copy on write. It may also contain information used to define the characteristics of the virtual machine associated with the images.

While VMDK, VHD, and QCOW are among the most commonly supported disk formats, they are far from universal. Some technologies still require raw file system images or other proprietary formats. The good news here is that conversion utilities exist for most formats available. If the desire is to have a single reference image that can be packaged for many different virtualization technologies, perhaps a raw file system or hard disk image is the best choice, as those can be directly converted to most other formats with little effort. Still, the question remains, why does this have to be so complicated? With a similar feature set among the most popular virtual disk formats, what is it that separates them? The difference lies in the metadata implementations. Perhaps in the future, common ground can be established and the disk format conversions will no longer be necessary.

2 Virtual Machine Configuration

When we look at what defines a virtual machine, there are many of the same pieces we find in stand-alone hardware. Along with the virtual disk or physical file systems, a virtual machine is allocated the basic resources required for a machine to function and be useful. This includes one or more virtual processors, a chunk of memory, and virtual or physical I/O devices. These are the core building blocks, and will differ among deployments of a given appliance based on customer requirements and available resources. In addition to these simple building blocks, there are typically a number of additional configuration options which help to define the virtual machine. These include migration and availability options, crash or debugging behavior, and console or terminal definitions. All of this configuration is specific to the actual deployment of an image, and should be defined within the hypervisor, controlling domain, or management infrastructure.

In addition to these site-specific configuration options, many virtualization vendors provide options for which kernel to boot, the initial ram disk or initrd image to be used, and other typical boot options. While there are specific circumstances where keeping such configuration options separate from the virtual machine container itself would be desirable, it is more frequently a

management headache for the consumer of virtual appliances. When the boot kernel is defined—or worse, located—outside of the central image, there is no simple way for an image to be self-maintained. While mechanisms exist for updating a virtual image similar to those for updating physical hosts, the guest does not typically have permission or ability to write to the host's file systems for such activities as updating the guest kernel or initrd. Simply put, any effective software appliance must be self-contained, and use standard tools for managing the contents of the appliance itself. Ideally the bootloader is also contained in the appliance image, but at the very minimum, a bootloader installed on the hypervisor should be able to read and interpret boot configuration information from within a guest image.

3 Paravirtualization vs. Full Virtualization

Both paravirtualized and fully virtualized machines have been around for quite some time, and each has distinct advantages. For the paravirtualized machine, the guest operating system is fully aware that it is operating under hypervisor control. The kernel has been modified to work directly with the hypervisor, and as much as possible to avoid instructions that are expensive to virtualize. The largest advantage to paravirtualization is performance. Generally speaking, a paravirtualized guest will outperform a fully virtualized guest on the same hardware, often by a substantial margin. With this being the case, why isn't every guest paravirtualized?

There are several obstacles to paravirtualization. The source level changes required to build a paravirtualized kernel can be large and invasive, in many cases tens of thousands of lines of code. These changes occur in core kernel code and can be much more complex than higher level driver code. While the nature of the changes required to support a given hypervisor can be very similar, the implementation details and ABI will vary from vendor to vendor, and even among versions of the hypervisor from the same vendor. It is not uncommon for a paravirtualization patch set to be several revisions behind the latest upstream kernel, or skip upstream revisions all together. This is unlikely to change until a given implementation has been accepted into the upstream kernel. The resources required to maintain such a large patch set outside of the upstream tree are considerable; maintaining the same code in the upstream kernel requires much fewer resources. There is also the small matter of guest operating systems which are not open source, or whose license does not allow the redistribution of changes. In these instances, paravirtualization is extremely difficult, if not impossible.

In the fully virtualized machine, the guest operating system does not need to know that it is being virtualized at all. The kernel operates exactly as it would on standard hardware. The hypervisor will trap necessary instructions and virtualize them without assistance from the guest. Standard devices such as network and block drivers are typically presented as virtual implementations of fairly well-supported physical devices to the guest so that no special drivers are needed. Modern CPUs include hardware support for virtualization, which improves performance and compatibility. While this method is an effective way to ensure compatibility with a large variety of guest operating systems, there is a high overhead in trapping all of the necessary instructions. To help with this performance problem, it is common for the hypervisor to support a number of paravirtualized device drivers. By replacing the common and well supported device drivers with new devices which are aware of the hypervisor, certain expensive instructions can be avoided and performance is improved dramatically. Typically, a virtualized guest with paravirtualized drivers will achieve performance much closer to that of a true paravirtualized guest.

This is another area of difficulty for the software appliance distributor. The vast number of virtualization vendors each have their own paravirtualized kernels, drivers, or guest tools. Some of these are open source, some are not. Regardless of source code availability, there is the question of which kernel versions these drivers will build against, or might be supported with. It is not uncommon for a vendor to have no working driver or tool set for two or three of the most recent upstream kernel versions, leaving them outside of the upstream stable support cycle all together. It is also possible that upstream kernel versions are skipped over entirely, making it difficult to find a common kernel version that can be supported by all of the desired virtualization targets that a software vendor might have. Luckily, it is quite possible to have a user space that supports a variety of kernel releases, ensuring that only the kernel version and associated virtualization drivers or tools are the only substantial changes between images. This leaves most of the QA and testing work intact, and still provides a substantial support savings over supporting entirely different general purpose operating systems.

It is hoped that some of these problems can be addressed generically. Changes to the Linux kernel are being made which make it possible to eventually build a single kernel which supports multiple virtualization solutions. Examples include `paravirt_ops` which shipped in the 2.6.20 kernel, and the VMI interface on top of `paravirt_ops` which is included in the 2.6.21 Linux kernel. While the initial groundwork for `paravirt_ops` and the VMI layer are present in mainline kernels, there is still a lot of work remaining to make them beneficial to the vast majority of users. In the short term, we have simply added another yet option for building virtualization solutions. Until stable releases of the majority players in virtualization have patches or products to support these new kernel interfaces available, and the older products are phased out, these interfaces simply represent one more option that must be supported. It really does have to get worse before it gets better.

Another proposal that has been floating around is a set of common paravirtualized drivers, which could be built as modules and provide many of the benefits associated with vendor provided tools and drivers while decreasing the number of configurations which must be built and supported. Unfortunately this proposal is in early stages and faces several obstacles. For instance, Xen provides xenbus instead of relying on the PCI specification for I/O virtualization. There is also the question of finding a common ground for block I/O, as many virtualization vendors have put considerable effort into optimizing block I/O for virtualized guests, and these implementations are not guaranteed to be compatible with one another. Still, if a agreement could be reached, the result would be basic paravirtualized drivers which could be maintained upstream, and present in the majority of Linux vendor kernels without overhead. Virtualization providers would still have the option of further optimization by using platform-specific drivers, but end users would see less basic overhead when using an image that for one reason or another could not easily deploy the platform-specific drivers.

4 Architectural incompatibility

Even when dealing with a single virtualization vendor, there are a few architectural anomalies to keep in mind. One particularly painful current example is PAE support. When dealing with 32-bit systems, both guest and host, there is a question of exactly how much memory is supported. In order for a 32-bit system to address more than 4GB of memory, PAE is supported on most modern x86 processors. In Linux, PAE is support is determined at kernel build time. Unfortunately a PAE-enabled kernel will not boot on physical or virtual hardware which does not actually support PAE mode. This is because PAE mode causes fairly significant changes to the page table structure regardless of the amount of actual memory in a system. This is important to know because several mainstream virtualization solutions take different approaches to PAE support. In the VMware case, PAE is supported on the host in modern versions, meaning the hypervisor can address more than 4GB of memory, but the guest does not support PAE mode even in instances where the host has more than 4GB available. While it is not a horrible limitation to say that a guest can only support less than 4GB of memory, it also means that a guest kernel cannot be built with PAE support and still work on all VMware deployments. (Whether PAE is supported in a VMware guest depends on the host kernel and on image-specific configuration settings.)

In the Xen case, the rules are less clear-cut. Xen has essentially three parts: the hypervisor, the domain 0 kernel, and the guest or unprivileged domain kernel. The hypervisor and the domain 0 kernel must always have matching PAE support, meaning if the domain 0 kernel is built with PAE support, the xen hypervisor must be built with PAE support as well. For guest domains, the situation is split between paravirtualized guests and hardware virtual machines using the hardware virtualization features of modern CPUs from Intel and AMD. A hardware virtualized machine can run with PAE either enabled or disabled, regardless of the domain 0 and hypervisor. For paravirtualized guest domains, the kernel must be built with the same PAE features of the hypervisor and domain 0. It is not possible to mix and match PAE between paravirtualized guests and the hypervisor with current releases of Xen. While it would be simple enough to say that PAE support should always be enabled, there are a few obstacles to this. Some hardware does not support PAE mode, particularly a large number of laptops with Intel Pentium M CPUs. Additionally, there are existing Xen hosts which do not support PAE for one reason or another. It is believed that over time non PAE implementations of 32-bit Xen will fall out of use, but the current issue is real and still somewhat common.

Guest architecture support will also vary according to

the hypervisor used. While many hypervisors currently available offer support for both 64-bit and 32-bit guests under a 64-bit hypervisor, several do not. Although the hardware transition is nearly complete (it is difficult to find mainstream desktops or servers which do not support x86_64 these days), it will still be some time before older 32-bit hardware is retired from use, and even longer before 32-bit applications are no longer supported by many vendors. This means that it may be necessary for software appliance vendors to offer both 32-bit and 64-bit guest images if they wish to ensure compatibility with the largest number of virtualization technologies. For applications which are only available in 32-bit flavors, it means that guests will have to run a 64-bit kernel in some circumstances, though a 32-bit user space is generally supported.

Conclusion

With well over a dozen virtualization solutions in use today, and more on the way, there is a lot of choice available to the consumer. Choice can be a double-edged sword. Competition drives innovation, we are seeing results from this at a rather fast pace today. Competition in the virtualization space also has the potential of driving support overhead to painful levels. Differing approaches to virtualization can ensure that the correct tool is available for any given job, but if the tool is too difficult to use, it is (more often than not) simply ignored in favor of the easier option.

Software vendors can leverage the benefits of virtual appliances now. While there are certainly obstacles to be overcome, they are not insurmountable. The advantages to a software appliance model are great, and the pains associated with this growth in virtualization technologies have to be addressed.

As developers, providers, and integrators of virtualization technology, we have to address these issues without allowing things to get out of hand. We need to look beyond the the technology itself, and see how it will be used. We need to make sure that the technology is consumable without a massive amount of effort from the consumers.

A New Network File System is Born: Comparison of SMB2, CIFS, and NFS

Steven M. French
IBM
Samba Team
sfrench@us.ibm.com

Abstract

In early 2007, SMB2 became the first widely deployed network file system protocol since NFS version 4. This presentation will compare it with its predecessors (CIFS and SMB) as well as with common alternatives. The strengths and weaknesses of SMB/CIFS (the most widely deployed network file system protocol) and NFS versions 3 and 4 (the next most popular protocols) and SMB2 will also be described.

Now that the CIFS POSIX Protocol extensions are implemented in the Linux kernel, Samba, and multiple operating systems, it is a good time to analyze whether SMB2 would be better for Linux comprared to CIFS POSIX Protocol extensions. In addition, alternatives such as HTTP, WebDav, and cluster file systems will be reviewed. Implementations of SMB2 are included in not just Vista and Longhorn, but also Samba client libraries and Wireshark (decoding support). Linux implementation progress and alternatives for SMB2 clients and servers will also be described along with recommendations for future work in this area.

1 Introduction

The SMB2 protocol, introduced in Microsoft Vista this year, is the default network file system on most new PCs. It differs from its predecessors in interesting ways.

Although a few experimental network file system protocols were developed earlier, the first to be widely deployed started in the mid-1980s: SMB (by IBM, Microsoft and others), AT&T's RFS protocol, AFS from Carnegie-Mellon University, NFS version 2—Sun [1] and Novell's NCP. The rapid increase in numbers of personal computers and engineering workstations quickly made network file systems an important mechanism for sharing programs and data. More than twenty years later, the successors to the ancient NFS and SMB protocols are still the default network file systems on almost all operating systems.

Even if HTTP were considered a network file system protocol, it is relatively recent, dating from the early 1990s, and its first RFC [RFC 1945] was dated May 1996. HTTP would clearly be a poor protocol for a general purpose network file system on most operating systems including Linux. Since HTTP lacked sufficient support for "distributed authoring" without locking operations, with little file metadata and lacking directory operations, "HTTP Extensions for Distributed Authoring—WEBDAV" (RFC 2518) was released in February 1999. WEBDAV did not, however, displace CIFS or NFS, and few operating systems have a usable in-kernel implementation of WEBDAV.

So after more than twenty years, despite the invention of some important cluster file systems and the explosion of interest in web servers, we are almost back where we started—comparing NFS [3] Version 4 with the current CIFS extensions and with a new SMB—the SMB2 protocol. File systems still matter. Network file systems are still critical in many small and large enterprises. File systems represent about 10% (almost 500KLOC) of the 2.6.21 Linux Kernel source code, and are among the most actively maintained and optimized components. The nfs[1] and cifs modules are among the larger in-kernel file systems.

Network file systems matter—the protocols that they depend on are more secure, full featured and much more

[1] lowercase "nfs" and "cifs" are used to refer to the implementation of the NFS and CIFS protocol (e.g. for Linux the `nfs.ko` and `cifs.ko` kernel modules), while uppercase "NFS" and "CIFS" refer to the network protocol.

complex than their ancestors. Some of the better NAS[2] implementations can perform as well as SAN and cluster file systems for key workloads.

2 Network File System Characteristics

Network protocols can be considered to be layered. Network file system protocols are the top layer—far removed from the physical devices such as Ethernet adapters that send bits over the wire. In the Open System Interconnection (OSI) model, network file system protocols would be considered as layer 6 and 7 ("Presentation" and "Application") protocols. Network file system protocols rely on lower level transport protocols (e.g. TCP) for reliable delivery of the network file systems protocol data units (PDUs), or include intermediate layers (as NFS has done with SunRPC) to ensure reliable delivery.

Network file system protocols share some fundamental characteristics that distinguish them from other "application level" protocols. Network file system clients and servers (and the closely related Network Attached Storage, NAS, servers) differ in key ways from cluster file systems and web browsers/servers:

- Files vs. Blocks or Objects: This distinction is easy to overlook when comparing network file system protocols with network block devices, cluster file systems and SANs. Network file systems read and write files not blocks of storage on a device. A file is more abstract—a container for a sequential series of bytes. A file is seekable. A file conventionally contains useful metadata such as ACLs or other security information, timestamps and size. Network file systems request data by file handle or filename or identifier, while cluster file systems operate on raw blocks of data. Network file system protocols are therefore more abstract, less sensitive to disk format, and can more easily leverage file ownership and security information.

- Network file system protocol operations match local file system entry points: Network file system protocol operations closely mirror the function layering of the file system layer (VFS) of the operating system on the client. Network file system operations on the wire often match one to one with the abstract VFS operations (read, write, open, close, create, rename, delete) required by the operating system. The OS/2 heritage of early SMB/CIFS implementations and the Solaris heritage of NFS are visible in a few network file system requests.

- Directory Hierarchy: Most network file systems assume a hierarchical namespace for file and directory objects and the directories that contain them.

- Server room vs. intranet vs. Internet: Modern network file system protocols have security and performance features that make them usable outside of the server room (while many cluster file systems are awkward to deploy securely across multiple sites). Despite this, HTTP and primitive FTP are still the most commonly used choices for file transfers over the Internet. Extensions to NFS version 4 and CIFS (DFS) allow construction of a global hierarchical namespace facilitating transparent failover and easier configuration.

- Application optimization: Because the pattern of network file system protocol requests often more closely matches the requests made by the application than would be the case for a SAN, and since the security and process context of most application requests can be easily determined, network file system servers and NAS servers can do interesting optimizations.

- Transparency: Network file systems attempt to provide local remote transparency so that local applications detect little or no difference between running over a network file system and a local file system.

- Heterogeneity: Network file system clients and servers are often implemented on quite different operating systems—clients access files without regard to their on-disk format. In most large enterprises, client machines running quite different operating systems access the same data on the same server at the same time. The CIFS (or NFS) network file system client that comes by default with their operating system neither knows nor cares about the operating system of the server. Samba server has been ported to dozens of operating systems, yet the server operating system is mostly

[2] Network Attached Storage (NAS) servers are closely related to network file servers.

transparent to SMB/CIFS clients. Network file systems are everywhere, yet are not always seen when running in multi-tier storage environments. They often provide consistent file access under large web servers or database servers or media servers. A network file system server such as Samba can easily export data on other network file systems, on removable media (CD or DVD), or on a local file system (ext3, XFS, JFS)—and with far more flexibility than is possible with most cluster file systems.

Network file systems differ in fundamental ways from web clients/servers and cluster file systems.

2.1 History of SMB Protocol

The SMB protocol was invented by Dr. Barry Feigenbaum of IBM's Boca Raton laboratory during the early development of personal computer operating system software. It was briefly named after his initials ("BAF") before changing the protocol name to "Server Message Block" or *SMB*. IBM published the initial SMB Specification book at the 1984 IBM PC Conference. A few years later a companion document, a detailed LAN Technical Reference for the NetBIOS protocol (which was used to transport SMB frames), was published. An alternative transport mechanism using TCP/IP rather than the Netbeui frames protocol was documented in RFCs 1001 and 1002 in 1987.

Microsoft, with early assistance from Intel and 3Com, periodically released documents describing new *dialects* of the SMB protocol. The LANMAN1.0 SMB dialect became the default SMB dialect used by OS/2. At least two other dialects were added to subsequent OS/2 versions.

In 1992, X/Open CAE Specification C209 provied better documentation for this increasingly important standard. The SMB protocol was not only the default network file system for DOS and Windows, but also for OS/2. IBM added Kerberos and Directory integration to the SMB protocol in its DCE DSS project in the early 1990s. A few years later Microsoft also added Kerberos security to their SMB security negotiation to their Windows 2000 products. Microsoft's Kerberos authentication encapsulated service tickets using SPNEGO in a new SMB SessionSetup variant, rather than using the original SecPkgX mechanism used by earlier SMB implementations (which had been documented by X/Open). The SMB protocol increasingly was used for purposes other than file serving, including remote server administration, network printing, networking messaging, locating network resources and security management. For these purposes, support for various network interprocess communication mechanisms was added to the SMB protocol including: Mailslots, Named Pipes, and the *LANMAN RPC*. Eventually more complex IPC mechanisms were built allowing encapsulating DCE/RPC traffic over SMB (even supporting complex object models such as DCOM).

In the mid 1990s, the SMBFS file system for Linux was developed. Leach and Naik authored various CIFS IETF Drafts in 1997, but soon CIFS Documentation activity moved to SNIA. Soon thereafter CIFS implementations were completed for various operating systems including OS/400 and HP/UX. The CIFS VFS for Linux was included in the Linux 2.6 kernel. After nearly four years, the SNIA CIFS Technical Reference [4] was released in 2002, and included not just Microsoft extensions to CIFS, but also CIFS Unix and Mac Extensions.

In 2003 an additional set of CIFS Unix Extensions was proposed, and Linux and Samba prototype implementations were begun. By 2005, Linux client and Samba server had added support for POSIX ACLs,[3] POSIX[4] path names, a request to return all information needed by statfs. Support for very large read requests and very large write responses was also added.

In April 2006, support for POSIX (rather than Windows-like) byte range lock semantics were added to the Samba server and Linux cifs client (Linux Kernel 2.6.17). Additional CIFS extensions were proposed to allow file I/O to be better POSIX compliant. In late 2006, and early 2007, joint work among four companies and the Samba team to define additional POSIX extensions to the CIFS protocol led to creation of a CIFS Unix Extensions wiki, as well as implementations of these new extensions [8]

[3] "POSIX ACLs" are not part of the official POSIX API. POSIX 1003.1e draft 17 was abandoned before standardization.

[4] In this paper, "POSIX" refers narrowly to the file API semantics that a POSIX-compliant operating system needs to implement. When the file system uses the CIFS network file system protocol, providing POSIX-like file API behavior to applications requires extensions to the CIFS network protocol. The CIFS "POSIX" Protocol Extensions are not part of the POSIX standard, rather a set of extensions to the network file system protocol to make it easier for network file system implementations to provide POSIX-like file API semantics.

in the Linux CIFS client and Samba server (Mac client and others in progress). The CIFS protocol continues to evolve, with security and clustering extensions among the suggestions for the next round of extensions. As the technical documentation of these extensions improves, more formal documentation is being considered.

2.2 History of NFS Protocol

NFS version 1 was not widely distributed, but NFS version 2 became popular in the 1980s, and was documented in RFC 1094 in 1989. Approximately 10 years after NFS version 2, NFS version 3 was developed. It was documented by Sun in RFC 1813 citerfc1813 in 1995. Eight years later RFC 3530 defined NFS version 4 (obsoleting the earlier RFC 3010, and completing a nearly five year standardization process). An extension to NFS version 3, "WebNFS," documented by Sun in 1996, attempted to show the performance advantages of a network file system for Internet file traffic in some workloads (over HTTP). The discussion of WebNFS increased the pressure on other network file systems to perform better over the Internet, and may have been a factor in the renaming of the SMB protocol—from "Server Message Block" to "Common Internet File System." Related to the work on NFS version 4 was an improvement to the SunRPC layer that NFS uses to transport its PDUs. The improved RPCSECGSS allowed support for Kerberos for authentication (as does CIFS), and allows negotiation of security features including whether to sign (for data integrity) or seal (for data privacy) all NFS traffic from a particular client to a particular server. The NFS working group is developing additional extensions to NFS (NFS version 4.1, pNFS, NFS over RDMA, and improvements to NFS's support for a global namespace).

The following shows new protocol operations introduced by NFS protocol versions 3 and 4:

NFS VERSION 2 Operations	
GETATTR	1
SETATTR	2
ROOT	3
LOOKUP	4
READLINK	5
WRITE	8
CREATE	9
REMOVE	10
RENAME	11
LINK	12
SYMLINK	13
MKDIR	14
RMDIR	15
READDIR	16
STATFS	17
New NFS VERSION 3 Operations	
ACCESS	4
READ	6
MKNOD	11
READDIRPLUS	17
FSSTAT	18
FSINFO	19
PATHCONF	20
COMMIT	21

New NFS Version 4 Operations	
CLOSE	4
DELEGPURGE	7
DELEGRETURN	8
GETFH	10
LOCK	12
LOCKT	13
LOCKU	14
LOOKUPP	16
NVERIFY	17
OPEN	18
OPENATTR	19
OPEN-CONFIRM	20
OPEN-DOWNGRADE	21
PUTFH	22
PUTPUBFH	23
PUTROOTFH	24
RENEW	30
RESTOREFH	31
SAVEFH	32
SECINFO	33
SETATTR	34
SETCLIENTID	35
SETCLIENTID-CONFIRM	36
VERIFY	37
RELEASE-LOCKOWNER	39

3 Current Network File System Alternatives

Today there are a variety of network file systems included in the Linux kernel, which support various protocols including: NFS, SMB/CIFS, NCP, AFS, and Plan9. In addition there are two cluster file systems now in the mainline Linux kernel: OCFS2 and GFS2. A few popular kernel cluster file systems for Linux that are not in mainline are Lustre and IBM's GPFS. The cifs and nfs file system clients for Linux are surprisingly similar in size (between 20 and 30 thousand lines of code) and change rate. The most common SMB/CIFS server for Linux is Samba, which is significantly larger than the Linux NFS server in size and scope. The most common Linux NFS server is of course nfsd, implemented substantially in kernel.

Windows Vista also includes support for various network file system protocols including SMB/CIFS, SMB2, and NFS.

4 SMB2 Under the Hood

The SMB2 protocol differs [7] from the SMB and CIFS protocols in the following ways:

- The SMB header is expanded to 64 bytes, and better aligned. This allows for increased limits on the

number of active connections (uid and tids) as well as the number of process ids (pids).

- The SMB header signature string is no longer 0xFF followed by "SMB" but rather 0xFE and then "SMB." In the early 1990s, LANtastic did a similar change in signature string (in that case from "SMB" to "SNB") to distinguish their requests from SMB requests.
- Most operations are handle based, leaving Create (Open/Create/OpenDirectory) as the only path based operation.
- Many redundant and/or obsolete commands have been eliminated.
- The file handle has been increased to 64 bits.
- Better support for symlinks has been added. Windows Services for Unix did not have native support for symlinks, but emulated them.
- Various improvements to DFS and other miscellaneous areas of the protocol that will become usable when new servers are available.
- "Durable file handles" [10] allowing easier reconnection after temporary network failure.
- Larger maximum operation sizes, and improved compound operation ("AndX") support also have been claimed but not proved.

Currently 19 SMB2 commands are known:

0x00	NegotiateProtocol	0x0A	Lock
0x01	SessionSetupAndX	0x0B	Ioctl
0x02	SessionLogoff	0x0C	Cancel
0x03	TreeConnect	0x0D	KeepAlive
0x04	TreeDisconnect	0x0E	Find
0x05	Create	0x0F	Notify
0x06	Close	0x10	GetInfo
0x07	Flush	0x11	SetInfo
0x08	Read	0x12	Break
0x09	Write		

Many of the infolevels used by the GetInfo/SetInfo commands will be familiar to those who have worked with CIFS.

5 POSIX Conformance

5.1 NFS

NFS version 3 defined 21 network file system operations (four more than NFS version 2) roughly corresponding to common VFS (Virtual File System) entry points that Unix-like operating systems require. NFS versions 2 and 3 were intended to be idempotent (stateless), and thus had difficulty preserving POSIX semantics. With the addition of a stateful lock daemon, an NFS version 3 client could achieve better application compatibility, but still can behave differently [6] than local file systems in at least four areas:

1. Rename of an open file. For example, the *silly rename* approach often used by nfs clients for renaming open files could cause `rm -rf` to fail.
2. Deleting an existing file or directory can appear to fail (as if the file were not present) if the request is retransmitted.
3. Byte range lock security (Since these services are distinct from the nfs server, both lockd and statd have had problems in this area).
4. write semantics (when caching was done on the client).

NFS also required additional protocol extensions to be able to support POSIX ACLs, and also lacked support for xattrs (OS/2 EAs), creation time (birth time), nanosecond timestamps, and certain file flags (immutable, append-only etc.). Confusingly, the NFS protocol lacked a file open and close operation until NFS version 4, and thus could only implement a weak cache consistency model.

5.2 NFSv4

NFS version 4, borrowing ideas from other protocols including CIFS, added support for an open and close operation, became stateful, added support for a rich ACL model similar to NTFS/CIFS ACLs, and added support for safe caching and a wide variety of extended attributes (additional file metadata). It is possible for an NFS version 4 implementation to achieve better application compatibility than before without necessarily sacrificing performance.

5.3 CIFS

The CIFS protocol can be used by a POSIX compliant operating system for most operations, but compensations are needed in order to properly handle POSIX locks, special files, and to determine approximate reasonable values for the mode and owner fields. There are other problematic operations that, although not strictly speaking POSIX issues, are important for a network file system in order to achieve true local remote transparency. They include symlink, statfs, POSIX ACL operations, xattrs, directory change notification (including inotify) and some commonly used ioctls (for example those used for the lsattr and chattr utilities). Without protocol extensions, the CIFS protocol can adequately be used for most important operations but differences are visible as seen in figure 1.

5.4 CIFS with Unix Protocol Extensions

As can be seen in figure 2, with the CIFS Unix Extensions it is possible to more accurately emulate local semantics for complex applications such as a Linux desktop.

The Unix Extensions to the CIFS Protocol have been improved in stages. An initial set, which included various new infolevels to TRANSACT2 commands in the range from 0x200 to 0x2FF (inclusive), was available when CAP_UNIX was included among the capabilities returned by the SMB negotiate protocol response.

Additional POSIX extensions are negotiated via a get and set capabilities request on the tree connection via a Unix QueryFSInfo and SetFSInfo level. Following is a list of the capabilties that may be negotiated currently:

- CIFS_UNIX_FCNTL_LOCKS_CAP
- CIFS_UNIX_POSIX_ACLS_CAP
- CIFS_UNIX_XATTR_CAP
- CIFS_UNIX_EXATTR_CAP
- CIFS_UNIX_POSIX_PATHNAMES_CAP (all except slash supported in pathnames)
- CIFS_UNIX_POSIX_PATH_OPS_CAP

A range of information levels above 0x200 has been reserved by Microsoft and the SNIA CIFS Working Group for Unix Extensions. These include Query/SetFileInformation and Query/SetPathInformation levels:

QUERY_FILE_UNIX_BASIC	0x200	Part of the initial Unix Extensions
QUERY_FILE_UNIX_LINK	0x201	Part of the initial Unix Extensions
QUERY_POSIX_ACL	0x204	Requires CIFS_UNIX_POSIX_ACL_CAP
QUERY_XATTR	0x205	Requires CIFS_UNIX_XATTR_CAP
QUERY_ATTR_FLAGS	0x206	Requires CIFS_UNIX_EXTATTR_CAP
QUERY_POSIX_PERMISSION	0x207	
QUERY_POSIX_LOCK	0x208	Requires CIFS_UNIX_FCNTL_CAP
SMB_POSIX_PATH_OPEN	0x209	Requires CIFS_UNIX_POSIX_PATH_OPS_CAP
SMB_POSIX_PATH_UNLINK	0x20a	Requires CIFS_UNIX_POSIX_PATH_OPS_CAP
SMB_QUERY_FILE_UNIX_INFO2	0x20b	Requires CIFS_UNIX_EXTATTR_CAP

Currently the CIFS Unix Extensions also include the following Query/SetFileSystemInformation levels that allow retrieving information about a particular mounted export ("tree connection"), and negotiating optional capabilities. Note that unlike NFS and SMB/CIFS, the CIFS Unix Extensions allow different capabilities to be negotiated in a more granular fashion, by "tree connection" rather than by server session.

If a server is exporting resources located on two very different file systems, this can be helpful.

SMB_QUERY_CIFS_UNIX_INFO	0x200	(Part of the original Unix Extensions)
SMB_QUERY_POSIX_FS_INFO	0x201	
SMB_QUERY_POSIX_WHO_AM_I	0x202	

These Unix Extensions allow a CIFS client to set and return fields such as uid, gid and mode, which otherwise have to be approximated based on CIFS ACLs. They also drastically reduce the number of network roundtrips and operations required for common path based operations. For example, with the older CIFS Unix Extensions, a file create operation takes many network operations: QueryPathInfo, NTCreateX, SetPathInfo, QueryPathInfo in order to implement local Unix create semantics correctly. File creation can be done in one network roundtip using the new SMB_POSIX_PATH_OPEN, which reduces latency and allows the server to better

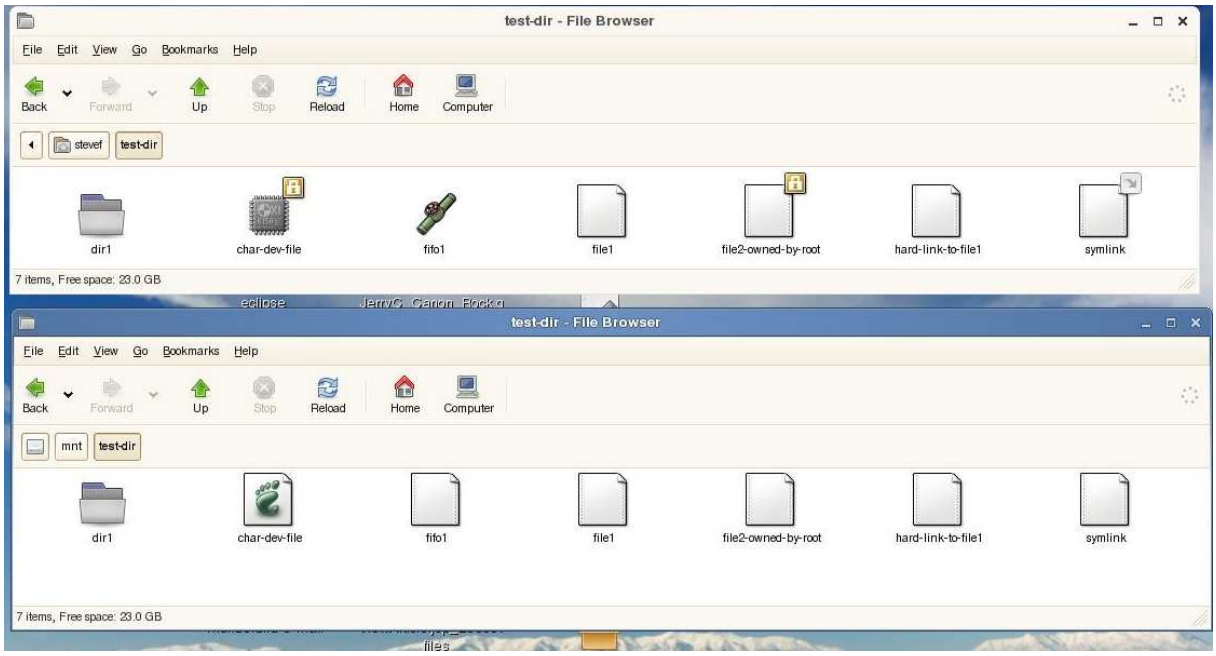


Figure 1: Without extensions to CIFS, local (upper window) vs. remote (below) transparency problems are easily visible

optimize. The improved atomicity of mkdir and create makes error handling easier (e.g. in case a server failed after a create operation, but before the SetPathInfo).

5.5 SMB2

The SMB2 protocol improves upon its predecessors by including symlink support. However, retrieving mode and Unix uid and gid from NTFS/CIFS ACLs is still awkward. SMB2 appears to be only slightly improved in this area, and substantially worse than the CIFS Unix Extensions for this purpose.

6 Performance

CIFS has often been described as a *chatty* protocol, implying that it is inherently slower than NFS, but this is misleading. Most of the chattiness observed in CIFS is the result of differences between the operating system implementations being compared (e.g. Windows vs. Linux). Another factor that leads to the accusation of the CIFS protocol being *chatty* (wasteful of network bandwidth) is due to periodic broadcast frames that contain server announcements (mostly in support of the Windows Network Neighborhood). These are not a required part of CIFS, but are commonly enabled on Windows servers so that clients and/or "Browse Masters" contain current lists of the active servers in a resource domain.

There are differences between these protocols that could significantly affect performance. Some examples include: compound operations, maximum read and write sizes, maximum number of concurrent operations, endian transformations, packet size, field alignment, difficult to handle operations, and incomplete operations that require expensive compensations.

To contrast features that would affect performance it is helpful to look at some examples.

6.1 Opening an existing file

The SMB2 implementation needs a surprising eight requests to handle this simple operation.

6.2 Creating a new file

The SMB2 protocol appears to match perfectly the requirements of the Windows client here. Attempting a simple operation like:

```
echo new file data > newfile
```

results in the minimum number of requests that would reasonably be expected (opencreate, write, close). Three requests and three responses (823 bytes total).

Figure 2: Better local (upper window) vs. remote (below) transparency with CIFS Unix extensions

6.3 Mount (NET USE)

Once again the SMB2 protocol appears to match well the requirement of the client with only 11 requests (four are caused by the Windows desktop trying to open `Desktop.ini` and `AutoRun.inf`).

7 Linux Implementation

Much of the progress on SMB2 has been due to excellent work by the Samba 4 team, led by Dr. Andrew Tridgell. Over the past year and a half, they have implemented a comprehensive client library for SMB2, implemented a test suite (not as comprehensive yet), implemented DCE/RPC over SMB2 (for remote administration), implemented a SMB2 server (not complete), and in cooperation with Ronnie Sahlberg, implemented a wireshark (ethereal) protocol analyzer.

8 Future Work and Conclusions

Although great progress has been made on a prototype user space client in Samba 4, an implementation of SMB2 in kernel on Linux also needs to be completed. We have started a prototype. The SMB2 protocol represents a modest improvement over the older SMB/CIFS protocol, and should be slightly better despite the slightly larger frame size caused by the larger header. With fewer commands to optimize and better aligned fields, performance may be slightly improved as server developers better tune their SMB2 implementations.

Despite the addition of support for symlinks, the SMB2 protocol lacks sufficient support for features needed by Unix and Linux clients. Adding Unix extensions to SMB2, similar to what has been done with CIFS, is possible and could reuse some of the existing Unix specific infolevels.

With current Linux kernels, NFS version 4 and CIFS (cifs client/Samba server) are good choices for network file systems for Linux to Linux. NFS performance for large file copy workloads is better, and NFS offers some security options that the Linux cifs client does not. In heterogeneous environments that include Windows clients and servers, Samba is often much easier to configure.

9 Acknowledgements

The author would like to express his appreciation to the Samba team, members of the SNIA CIFS technical work

octet 1	2	3	4	5	6	7	8
RFC 1001 msg type (session)	SMB length (some reserve top 7 bits)			0xFF	'S'	'M'	'B'
SMB Command	Status (error) code				SMB flags	SMB flags2	
Process ID (high order)	SMB Signature						
SMB signature (continued)	Reserved			Tree Identifier		Process Id (Low)	
SMB User Identifier	Word Count	(variable number of 16 bit parameters follow)			Byte Count (size of data area)		(data area follows)

Table 1: SMB Header Format (39 bytes + size of command specific wct area)

octet 1	2	3	4	5	6	7	8
RFC 1001 msg type (session)	SMB length			0xFE	'S'	'M'	'B'
SMB Header length (64)	reserved			Status (error) code			
SMB2 Command	Unknown			SMB2 Flags			
Reserved				Sequence number			
Sequence Number (continued)				Process Id			
Tree Identifier				SMB User Identifier			
SMB User Identifier				SMB Signature			
SMB Signature (continued)							
SMB Signature (continued)				SMB2 Parameter length (in bytes)		Variable length SMB Parm	Variable length SMB Data

Table 2: SMB2 Header Format (usually 68 bytes + size of command specific parameter area)

octet 1	2	3	4	5	6	7	8
SunRPC Fragment Header				XID			
Message Type (Request vs. Response)				SunRPC Version			
Program: NFS (100003)				Program Version (e.g. 3)			
NFS Command				Authentication Flavor (e.g. AUTH_UNIX)			
Credential Length				Credential Stamp			
Machine Name length				Machine name (variable size)			
Machine Name (continued, variable length)							
Unix UID				Unix GID			
Auxiliary GIDs (can be much larger)							
Verifier Flavor				Verifier Length			
NFS Command Parameters and/or Data follow							

Table 3: SunRPC/NFSv3 request header format (usually more than 72 bytes + size of nfs command)

group, and others in analyzing and documenting the SMB/CIFS protocol and related protocols so well over the years. This is no easy task. In addition, thanks to the Wireshark team and Tridge for helping the world understand the SMB2 protocol better, and of course thanks to the Linux NFSv4 developers and the NFS RFC authors, for implementing and documenting such a complex protocol. Thanks to Dr. Mark French for pointing out some of the many grammar errors that slipped through.

10 Legal Statement

References

[1] Sun Microsystems Inc. RFC 1094: NFS: Network File System Protocol Specification. March 1989. See `http://www.ietf.org/rfc/rfc1094.txt`

[2] Callaghan et al. RFC 1813: NFS Version 3 Protocol Specification. June 1995. See `http://www.ietf.org/rfc/rfc1813.txt`

[3] S. Shepler, et al. RFC 3530: Network File System (NFS) version 4 Protocol. April 2003. See `http://www.ietf.org/rfc/rfc3530.txt`

[4] J. Norton, et al. SNIA CIFS Technical Reference. March 2002. See `http://www.snia.org/tech_activities/CIFS/CIFS-TR-1p00_FINAL.pdf`

[5] C. Hertel. Implementing CIFS. 2004. See `http://ubiqx.org/cifs/`

[6] O. Kirch. Why NFS Sucks. *Proceedings of the 2006 Ottawa Linux Symposium*, Ottawa, Canada, July 2006.

[7] Dr. A. Tridgell. Exploring the SMB2 Protocol. SNIA Storage Developer Conference. September 2006. `http://samba.org/~tridge/smb2.pdf`

[8] CIFS Unix Extensions. `http://wiki.samba.org/index.php/UNIX_Extensions`

[9] Linux CIFS Client and Documentation. `http://linux-cifs.samba.org`

[10] What's new in SMB in Windows Vista `http://blogs.msdn.com/chkdsk/archive/2006/03/10/548787.aspx`

Supporting the Allocation of Large Contiguous Regions of Memory

Mel Gorman
IBM and University of Limerick
mel@csn.ul.ie

Andy Whitcroft
IBM
andyw@uk.ibm.com

Abstract

Some modern processors such as later Opterons® and Power® processors are able to support large pages sizes such as 1GiB and 16GiB. These page sizes are impractical to reserve at boot time because of the amount of memory that is potentially wasted. Currently, Linux® as it stands is not well suited to support multiple page sizes because it makes no effort to satisfy allocations for contiguous regions of memory. This paper will discuss features under development that aim to support the allocation of large contiguous areas.

This paper begins by discussing the current status of mechanisms to reduce external fragmentation in the page allocator. The reduction of external fragmentation results in sparsely populated superpages that must be reclaimed for contiguous allocations to succeed. We describe how poor reclaim decisions offset the performance benefits of superpages in low-memory situations, before introducing a mechanism for the intelligent reclaim of contiguous regions. Following a presentation of metrics used to evaluate the features and the results, we propose a memory compaction mechanism that migrates pages from sparsely populated to dense regions when enough memory is free to avoid reclaiming pages. We conclude by highlighting that parallel allocators prevent contiguous allocations by taking free pages from regions being reclaimed. We propose a method for addressing this by making pages temporarily unavailable to allocators.

1 Introduction

Any architecture supporting virtual memory is required to map virtual addresses to physical addresses through an address translation mechanism. Recent translations are stored in a cache called a *Translation Lookaside Buffer (TLB)*. *TLB Coverage* is defined as memory addressable through this cache without having to access the master tables in main memory. When the master table is used to resolve a translation, a *TLB Miss* is incurred. This can have as significant an impact on *Clock cycles Per Instruction (CPI)* as CPU cache misses [3]. To compound the problem, the percentage of memory covered by the TLB has decreased from about 10% of physical memory in early machines to approximately 0.01% today [6]. As a means of alleviating this, modern processors support multiple page sizes, usually up to several megabytes, but gigabyte pages are also possible. The downside is that processors commonly require that physical memory for a page entry be contiguous.

In this paper, mechanisms that improve the success rates of large contiguous allocations in the Linux kernel are discussed. Linux already supports two page sizes, referred to as the *base page* and the *huge page*. The paper begins with an update on previous work related to the placement of pages based on their mobility [2]. A percentage of pages allocated by the kernel are movable due to the data being referenced by page tables or trivially discarded. By grouping these pages together, contiguous areas will be moved or reclaimed to satisfy large contiguous allocations. As a bonus, pages used by the kernel are grouped in areas addressable by fewer large TLB entries, reducing TLB misses.

Work on the intelligent reclaim of contiguous areas is then discussed. The regular reclaim algorithm reclaims base pages but has no awareness of contiguous areas. Poor page selections result in higher latencies and lower success rates when allocating contiguous regions.

Metrics are introduced that evaluate the placement policy and the modified reclaim algorithm. After describing the test scenario, it is shown how the placement policy keeps a percentage of memory usable for contiguous allocations, and the performance impact is discussed. It is then shown how the reclaim algorithm satisfies the allocation of contiguous pages faster than the regular reclaim.

The remainder of the paper proposes future features to improve the allocation of large contiguous regions. Investigation showed that contiguous areas were sparsely populated and that compacting memory would be a logical step. A memory compaction mechanism is proposed that will be rarely triggered, as an effective placement policy significantly reduces the requirement for compaction [1] [4]. Additional investigation revealed that parallel allocators use pages being reclaimed, preempting the contiguous allocation. This is called the *racing allocator* problem. It is proposed that pages being reclaimed be made unavailable to other allocators.

2 External Fragmentation

External fragmentation refers to the inability to satisfy an allocation because a sufficiently sized free area does not exist despite enough memory being free overall [7]. Linux deals with external fragmentation by rarely requiring contiguous pages. This is unsuitable for large, contiguous allocations. In our earlier work, we defined metrics for measuring external fragmentation and two mechanisms for reducing it. This section will discuss the current status of the work to reduce external fragmentation. It also covers how page types are distributed throughout the physical address space.

2.1 Grouping Pages By Mobility

Previously, we grouped pages based on their ability to be reclaimed and called the mechanism *anti-fragmentation*. Pages are now grouped based on their ability to be moved, and this is called *grouping pages by mobility*. This takes into account the fact that pages `mlock()`ed in memory are movable by page migration, but are not reclaimable.

The standard buddy allocator always uses the smallest possible block for an allocation, and a minimum number of pages are kept free. These free pages tend to remain as contiguous areas until memory pressure forces a split. This allows occasional short-lived, high-order allocations to succeed, which is why setting `min_free_kbytes` to 16384[1] benefits Ethernet cards using jumbo frames. The downside is that once split, there is no guarantee that the pages will be freed as a contiguous area. When grouping by mobility, the minimum number of free pages in a zone are stored in contiguous areas. Depending on the value of `min_free_kbytes`, a number of areas are marked `RESERVE`. Pages from these areas are allocated when the alternative is to fail.

Previously, bits from `page→flags` were used to track individual pages. A bitmap now records the mobility type of pages within a `MAX_ORDER_NR_PAGES` area. This bitmap is stored as part of the `struct zone` for all memory models except the `SPARSEMEM`, where the memory section is used. As well as avoiding the consumption of `page→flags` bits, tracking page type by area controls fragmentation better.

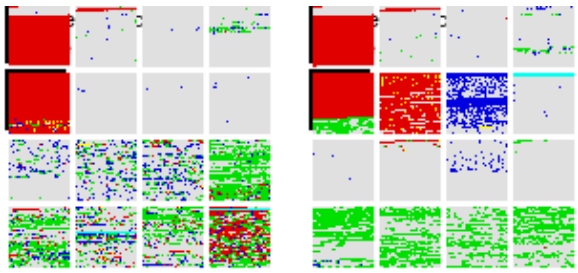
Previously effective control of external fragmentation required that `min_free_kbytes` be 5% to 10% of physical memory, but this is no longer necessary. When it is known there will be bursts of high-order atomic allocations during the lifetime of the system, `min_free_kbytes` should be increased. If it is found that high order allocations are failing, increasing `min_free_kbytes` will free pages as contiguous blocks over time.

2.2 Partitioning Memory

When grouping pages by mobility, the maximum number of superpages that may be allocated on demand is dependent on the workload and the number of movable pages in use. This level of uncertainty is not always desirable. To have a known percentage of memory available as contiguous areas, we create a zone called `ZONE_MOVABLE` that only contains pages that may be migrated or reclaimed. Superpage allocations are not movable or reclaimable but if a **sysctl** is set, superpage allocations are allowed to use the zone. Once the zone is sized, there is a reasonable expectation that the superpage pool can be grown at run-time to at least the size of `ZONE_MOVABLE` while leaving the pages available for ordinary allocations if the superpages are not required.

The size in bytes of the `MAX_ORDER` area varies between systems. On x86 and x86-64 machines, it is 4MiB of data. On PPC64, it is 16MiB; on an IA-64 supporting huge pages, it is often 1GiB. If a developer requires a 1GiB superpage on Power there is no means to providing it, as the buddy allocator does not have the necessary free-lists. Larger allocations could be satisfied by increasing `MAX_ORDER`, but this is a compile-time change and not desirable in the majority of cases. Generally, supporting allocations larger than `MAX_ORDER_`

[1] A common recommendation when using jumbo frames.



Standard Allocator Grouping By Mobility

Figure 1: Distribution of Page Types

NR_PAGES requires adjacent MAX_ORDER_NR_PAGES areas to be the same mobility type. The memory partition provides this guarantee.

The patches to group pages by mobility and partition memory were merged for wider testing in the Linux kernel version 2.6.21-rc2-mm2 despite some scepticism on the wider utility of the patches. Patches to allocate adjacent MAX_ORDER_NR_PAGES are planned.

2.3 Comparing Distribution

Figure 1 shows where pages of different mobility types are placed with the standard allocator and when grouping by mobility. Different colours are assigned to pages of different mobility types and each square represents MAX_ORDER_NR_PAGES. When the snapshot was taken the system had been booted, a 32MiB file was created and then deleted. It is clear in the figure that MAX_ORDER_NR_PAGES areas contain pages of mixed types with the standard allocator, but the same type when when grouping by mobility. This "mixing" in the standard allocator means that contiguous allocations are not likely to succeed even if all reclaimable memory is released.

It is clear from the figure that grouping pages by mobility does not guarantee that all of memory may be allocated as contiguous areas. The algorithm depends on a number of areas being marked MOVABLE so that they may be migrated or reclaimed. To a lesser extent it depends on RECLAIMABLE blocks being reclaimed. Reclaiming those blocks is a drastic step because there is no means to target the reclamation of kernel objects in a specific area.

Despite these caveats, the result when grouping pages by mobility is that a high percentage of memory may be allocated as contiguous blocks. With memory partitioning, a known percentage of memory will be available.

3 Reclaim

When an allocation request cannot be satisfied from the free pool, memory must be reclaimed or compacted. Reclaim is triggered when the free memory drops below a watermark, activating **kswapd**, or when available free memory is so low that memory is reclaimed directly by a process.

The allocator is optimised for base page-sized allocations, but the system generates requests for higher order areas. The regular reclaim algorithm fares badly in the face of such requests, evicting significant portions of memory before areas of the requested size become free. This is a result of the *Least Recently Used (LRU)* reclaim policy. It evicts pages based on age without taking page locality into account.

Assuming a random page placement at allocation and random references over time, then pages of similar LRU age are scattered throughout the physical memory space. Reclaiming based on age will release pages in random areas. To guarantee the eviction of two adjacent pages requires 50% of all pages in memory to be reclaimed. This requirement increases with the size of the requested area tending quickly towards 100%. Awareness is required of the size and alignment of the allocation or the performance benefits of superpage use are offset by the cost of unnecessarily reclaiming memory.

3.1 Linear Area Reclaim

To fulfil a large memory request by reclaiming, a contiguous aligned area of pages must be evicted. The resultant free area is then returned to the requesting process. Simplistically this can be achieved by linearly scanning memory, applying reclaim to suitable contiguous areas. As reclaim completes, pages coalesce into a contiguous area suitable for allocation.

The linear scan of memory ignores the age of the page in the LRU lists. But as previously discussed, scanning based on page age is highly unlikely to yield a contiguous area of the required size. Instead, a hybrid of these two approaches is used.

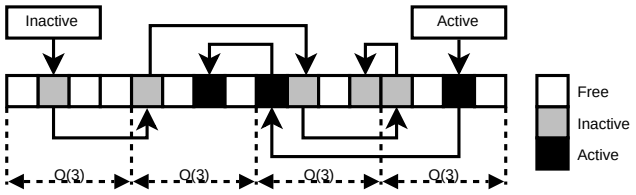


Figure 2: Area Reclaim area selection

3.2 LRU-Biased Area Reclaim

LRU-Biased Area Reclaim (Area Reclaim)[2] is a hybrid reclaim algorithm incorporating linear area reclaim into the regular LRU-based aging algorithm. Instead of linearly scanning memory for a suitable area, the tails of the active and inactive LRU lists are used as a starting point.

Area Reclaim follows the regular approach of targeting pages in the LRU list order. First, an area in the active list is selected based on the oldest page in that list. Active pages in that area are rotated to the inactive list. An area in the inactive list is then selected based on the oldest page in that list. All pages on the LRU lists within this area are then reclaimed. When there is allocation pressure at a higher order, this tends to push groups of pages in the same area from the active list onto the head of the inactive list increasing the chances of reclaiming areas at its tail in the future. On the assumption there will be future allocations of the same size, **kswapd** applies pressure at the largest size required by an in-progress allocation.

Figure 2 shows an example memory layout. As an example, consider the application of Area Reclaim at order-3. The active and inactive lists work their way through the pages in memory indicating the LRU-age of those pages. In this example, the end of the active list is in the second area and the end of the inactive list is in the third area. Area Reclaim first applies pressure to the active list, targeting the second area. Next it applies pressure to the inactive list, targeting area three. All pages in this area are reclaimable or free and it will coelesce.

While targeting pages in an area, we maintain the age-based ordering of the LRU to some degree. The downside is that younger, active and referenced pages in the same area are all targeted prematurely. The higher the order of allocation, the more pages that are unfairly treated. This is unavoidable.

[2]This was initially based on Peter Zijlstra's modifications to last year's Linear Reclaim algorithm, and is commonly known as Lumpy Reclaim.

x86-64 Test Machine

CPU	Opteron® 2GHz
# Physical CPUs	2
# CPUs	4
Main Memory	1024MiB

PPC64 Test Machine

CPU	Power5® PPC64 1.9GHz
# Physical CPUs	2
# CPUs	4
Main Memory	4019MiB

Figure 3: Specification of Test Machines

4 Experimental Methodology

The mechanisms are evaluated using three tests: one related to performance, and two that are known to externally fragment the system under normal conditions. Each of the tests is run in order, without intervening reboots, to maximise the chances of the system being fragmented. The tests are as follows.

kernbench extracts the kernel and then builds it three times. The number of jobs **make** runs is the number of processors multiplied by two. The test gives an overall view of the kernel's performance and is sensitive to changes that alter scalability.

HugeTLB-Capability is unaltered from our previous study. For every 250MiB of physical memory, a kernel compile is executed in parallel. Attempts are made to allocate hugepages through the kernel's `proc` interface under load and at the end of test.

Highalloc-Stress builds kernels in parallel, but in addition, **updatedb** runs so that the kernel will make many small unmovable allocations. Persistent attempts are made to allocate hugepages at a constant rate such that **kswapd** should not queue more than 16MiB/s I/O. Previous tests placed 300MiB of data on the I/O queue in the first three seconds, making comparisons of reclaim algorithms inconclusive.

All benchmarks were run using driver scripts from VM-Regress 0.80[3] in conjunction with the system that generates the reports on `http://test.kernel.org`. Two machines were used to run the benchmarks based on 64-bit AMD® Opteron and POWER5® architectures, as shown in Figure 3.

On both machines, a minimum free reserve of $5 * MAX\_ORDER\_NR\_PAGES$ was set, representing 2% of physical memory. The placement policy is effective with a lower reserve, but this gives the most predictable results. This is due to the low frequency that page types are mixed under memory pressure. In contrast, our previous study required fives times more space to reduce the frequency of fallbacks.

5 Metrics

In this section, five metrics are defined that evaluate fragmentation reduction and the modified reclaim algorithm. The first metric is *system performance*, used to evaluate the overhead incurred when grouping pages by mobility. The second metric is overall allocator *effectiveness*, measuring the ability to service allocations. Effectiveness is also used as the basis of our third metric, *reclaim fitness*, measuring the correctness of the reclaim algorithm. The fourth metric is *reclaim cost*, measuring the processor and I/O overheads from reclaim. The final metric is *inter-allocation latency*, measuring how long it takes for an allocation to succeed.

5.1 System Performance

The performance of the kernel memory allocator is critical to the overall system. Grouping pages by mobility affects this critical path and any significant degradation is intolerable. The **kernbench** benchmark causes high load on the system, particularly in the memory allocator. Three compilation runs are averaged, giving processor utilisation figures to evaluate any impact of the allocator paths.

5.2 Effectiveness

The effectiveness metric measures what percentage of physical memory can be allocated as superpages

$$E = (A_r * 100)/A_t$$

where A_r is the number of superpages allocated and A_t is the total number of superpages in the system. During the **Highalloc-Stress** test, attempts are made to allocate superpages under load and at rest, and the effectivess is measured. Grouping pages by mobility should show an increase in the effectiveness when the system is at rest at the end of a test. Under load, the metric is a key indicator of the intelligent reclaim algorithm's area selection.

5.3 Reclaim Fitness

Reclaim fitness validates the reclaim algorithm. When applied to 100% of memory, any correct algorithm will achieve approximately the same effectiveness. A low variance implies a correct algorithm. An incorrect algorithm may prematurely exit or fail to find pages.

5.4 Reclaim Cost

The reclaim cost metric measures the overhead incurred by scanning, unmapping pages from processes, and swapping out. Any modification to the reclaim algorithm affects the overall cost to the system. Cost is defined as:

$$C = log_{10}((C_s * W_s) + (C_u * W_u) + (C_w * W_w))$$

where C_s is the number of pages scanned, C_u is the number of pages unmapped, and C_w is the number of pages we have to perform I/O for in order to release them. The scaling factors are chosen to give a realistic ratio of each of these operations. W_s is given a weight of 1, W_u is weighted as 1000 and W_w is weighted at 1,000,000.

The cost metric is calculated by instrumenting the kernel to report the scan, unmap, and write-out rates. These are collected while testing effectiveness. This metric gives a measure of the cost to the system when reclaiming pages by indicating how much additional load the system experiences as a result of reclaim.

[3] `http://www.csn.ul.ie/~mel/projects/vmregress/vmregress-0.80.tar.gz`

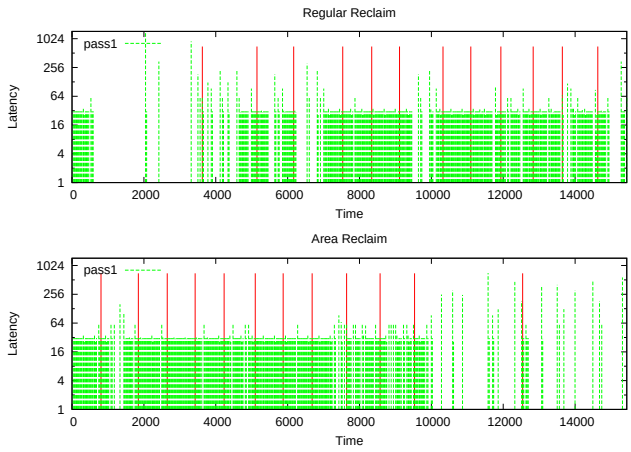


Figure 4: Inter-Allocation Latencies

x86-64 Test Machine

CPU Time	Mobility		Delta (%)
	Off	On	
User	85.83	86.78	1.10
System	35.92	34.07	–5.17
Total	121.76	120.84	–0.75

PPC64 Test Machine

CPU Time	Mobility		Delta (%)
	Off	On	
User	312.43	312.24	–0.06
System	16.89	17.24	2.05
Total	329.32	329.48	0.05

Figure 5: Mobility Performance Comparison

5.5 Inter-Allocation Latency

Given a regular stream of allocation attempts of the same size, the inter-allocation latency metric measures the time between successful allocations. There are three parts to the metric. The inter-allocation latency variability is defined as the standard deviation of the inter-allocation delay between successful allocations. The mean is the arithmetic mean of these inter-allocation delays. The worst-case allocation time is simply the worst inter-allocation delay.

The inter-allocation times are recorded when measuring effectivess. Figure 4 shows the raw inter-allocation times for regular and Area Reclaim. The fixed-height vertical red lines indicate where an additional 5% of memory was allocated as superpages. A vertical green bar indicates when an allocation succeeded and the height indicates how long since the last success. The large portions of white on the left side of the regular reclaim graph indicate the time required for enough memory to be reclaimed for contiguous allocations to succeed. In contrast, Area Reclaim does not regularly fail until 50% of memory is allocated as superpages.

A key feature of a good reclaim algorithm is to provide a page of the requested size in a timely manner. The inter-allocation variability metric gives us a measure of the consistency of the algorithm. This is particularly significant during the transition from low to high allocator load, as allocations are time critical.

6 Results

Four different scenarios were tested: two sets of runs evaluating the effectiveness and performance of the page mobility changes, and two sets of runs evaluating the Regular and Area Reclaim algorithms.

Figure 5 shows the elapsed processor times when running **kernbench**. This is a `fork()`-, `exec()`-, I/O-, and processor-intensive workload and a heavy user of the kernel page allocator, making it suitable for measuring performance regressions there. On x86-64, there is a significant and measurable improvement in system time and overall processor time despite the additional work required by the placement policy. In Linux, the kernel address space is a linear mapping of physical memory using superpage *Page Table Entries (PTEs)*. Grouping pages by mobility keeps kernel-related allocations together in the same superpage area, reducing TLB misses in the kernel portion of the working set. A performance gain is found on machines where the kernel portion of the working set exceeds TLB reach when the kernel allocations are mixed with other allocation types.

In contrast, the PPC64 figures show small performance regressions. Here, the kernel portion of the address space is backed by superpage entries, but the PPC64 processor has at least 1024 TLB entries, or almost ten times the number of x86-64. In this case, the TLB is able to hold the working set of the kernel portion whether pages are grouped by mobility or not. This is compounded by running **kernbench** immediately after boot, causing allocated physical pages to be grouped together. PPC64 is expected to show similar improvements due

to reduced TLB misses with workloads that have large portions of their working sets in the kernel address space and when the system has been running for a long time.

Where there are no benefits due to reduced TLB misses in the kernel portion of a working set, regressions of between 0.1% and 2.1% are observed in **kernbench**. However, the worst observed regression in overall processor time is 0.12%. Given that **kernbench** is an unusually heavy user of the kernel page allocator and that superpages potentially offer considerable performance benefits, this minor slowdown is acceptable.

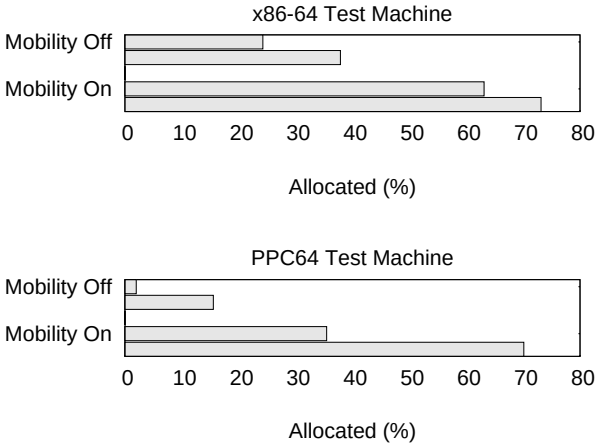


Figure 6: Mobility Effectiveness

Figure 6 shows overall effectiveness. The pair of bars show the percentage of memory successfully allocated as superpages under load during the **Highalloc-Stress** stress and at the end when the system is at rest. The figures show that grouping the pages significantly improves the success rates for allocations. In particular on PPC64, dramatically fewer superpages were available at the end of the tests in the standard allocator. It has been observed that the longer systems run without page mobility enabled, the closer to 0% the success rates are.

Figure 7 compares the percentage of memory allocated as huge pages at the end of all test for both reclaim algorithms. Both algorithms should be dumping almost all of memory, so the figures should always be comparable. The figures are comparable or improved, so it is known that Area Reclaim will find all reclaimable pages when under pressure. This validates the Area Reclaim scanner.

Arch	Reclaim		Delta (%)
	Regular	Area	
x86-64	64.53	73.15	8.62
PPC64	72.11	70.12	-1.99

Figure 7: Correctness

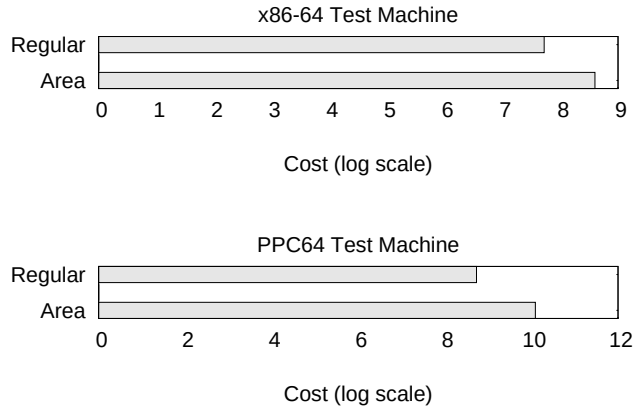


Figure 8: Cost Comparison

Figure 8 shows that the cost of Area Reclaim are higher than those of Regular reclaim, but not by a considerable margin. Later we will show the the time taken to allocate the required areas is much lower with Area Reclaim and justifies the cost. Cost is a trade-off. On one hand the algorithm must be effective at reclaiming areas of the requested size in a reasonable time. On the other, it must avoid adversely affecting overall system utility while it is in operation.

The first row of graphs in Figure 9 shows the inter-allocation variability as it changes over the course of the test. Being able to deliver ten areas 20 seconds after they are requested is typically of no use. Note that in both algorithms, the variability is worse towards the start, with Area Reclaim significantly out-performing Regular Reclaim.

The second row of graphs in Figure 9 shows the mean inter-allocation latency as it changes over the course of the test. It is very clear that the Area Reclaim inter-allocation latency is more consistent for longer, and generally lower than that for Regular reclaim.

The third row of graphs in Figure 9 shows the worst-case inter-allocation latency. Although the actual worst-case latencies are very similar with the two algorithms, it is clear that Area Reclaim maintains lower latency for much longer. The worst-case latencies for Area Reclaim are at the end of the test, when very few potential contiguous regions exist.

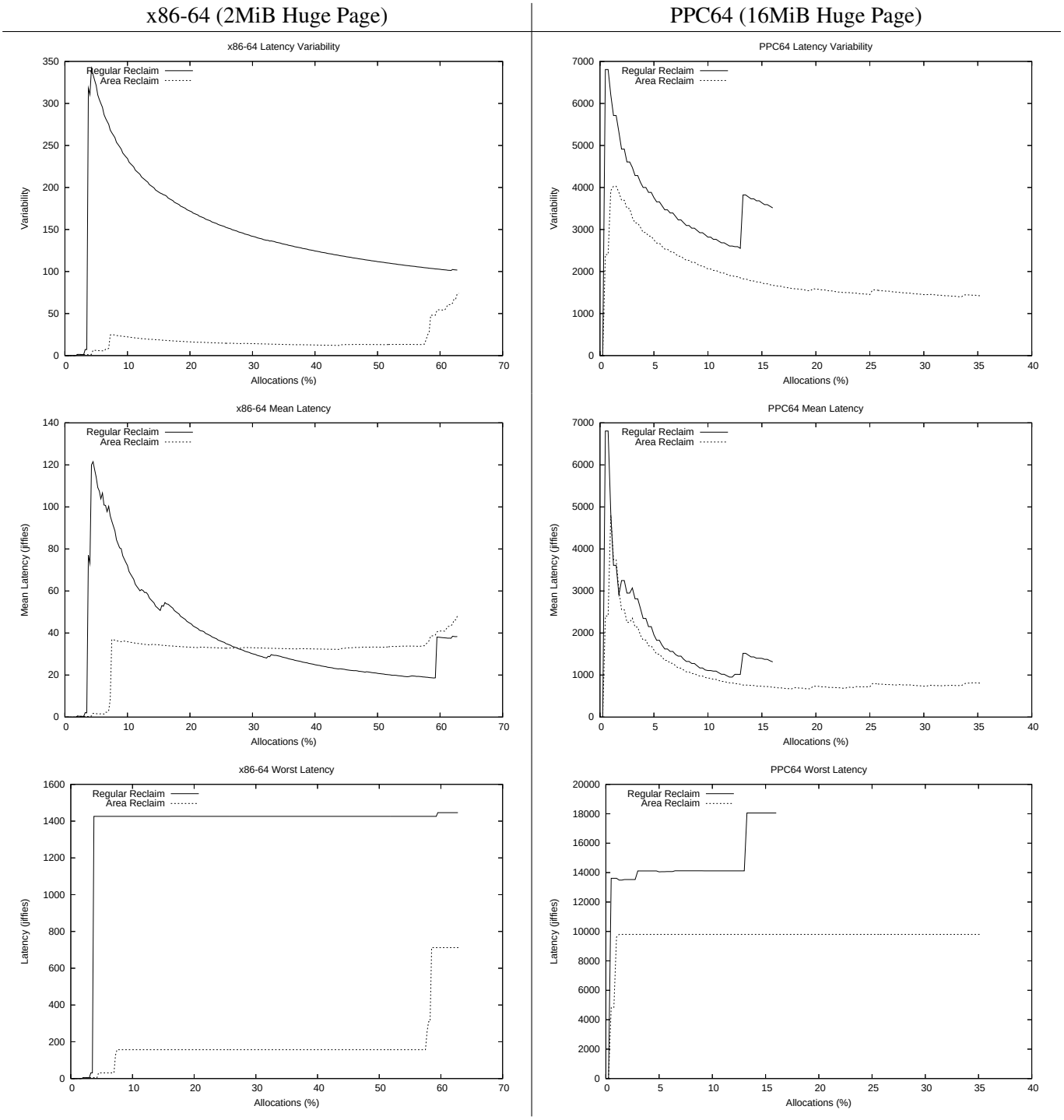


Figure 9: Latency Comparison

The superpage size on PPC64 is eight times larger than that on x86-64. Increased latencies between the architectures is to be expected, as reclaiming the required contiguous area is exponentially more difficult. Inspection of the graphs shows that Area Reclaim allocates superpages faster on both x86-64 and PPC64. The increased effectiveness of the Area Reclaim algorithm is particularly noticeable on PPC64 due to the larger superpage size.

6.1 Results Summary

The effectiveness and performance comparisons show that grouping pages by mobility and Area Reclaim is considerably better at allocating contiguous regions than the standard kernel, while still performing well. The cost and latency metrics show that for a moderate increase in work, we get a major improvement in allocation latency under load. A key observation is that early allocations under load with Area Reclaim have considerably lower latency and variability in comparison to the standard allocator. Typically a memory consumer will not allocate all of memory as contiguous areas, but allocate small numbers of contiguous areas in short bursts. Even though it is not possible to anticipate bursts of contiguous allocations, they are handled successfully and with low latency by the combination of grouping pages by mobility and Area Reclaim.

7 Memory Compaction

Reclaim is an expensive operation and the cost might exceeed the benefits of using superpages, particularly when there is enough memory free overall to satisfy the allocation. This paper proposes a memory compaction mechanism that moves pages so that there are fewer, but larger and contiguous, free areas. The term defragmentation is avoided because it implies all pages are movable, which is not the case in Linux. At the time of writing, no such memory compaction daemon has been developed, although a page migration mechanism[5] does exist in the kernel.[4]

7.1 Compaction Mechanism

The compaction mechanism will use the existing page migration mechanism to move pages. The page mobility type information stored for a `MAX_ORDER_NR_PAGES` area will be used to select where pages should be migrated. Intuitively the best strategy is to move pages from sparsely to densely populated areas. This is unsuitable for two reasons. First, it involves a full scan of all pages in a zone and sorting areas based on density. Second, when compaction starts, areas that were previously sparse may no longer be sparse unless the system was frozen during compaction, which is unacceptable.

The compaction mechanism will group movable pages towards the end of a zone. When grouping pages by mobility, the location of unmovable pages is biased towards the lower addresses, so these strategies work in conjunction. As well as supporting the allocation of very large contiguous areas, biasing the location of movable pages towards the end of the zone potentially benefits the hot-removal of memory and simplifies scanning for free pages to migrate to.

When selecting free pages, the *free page scanner* begins its search at the end of a zone and moves towards the start. Areas marked `MOVABLE` are selected. The free pages contained within are removed from the free-lists and stored on a private list. No further scanning for free pages occurs until the pages on the private list are depleted.

When selecting pages to move, the *migration scanner* searches from the start of a zone and moves towards the end. It searches for pages that are on the LRU lists, as these are highly likely to be migratable. The selection of an area is different when a process or a compaction daemon is scanning. This process will be described in the next two sections.

A compaction run always ends when the two scanners meet. At that point, it is known that it is very unlikely that memory can be further compacted unless memory is reclaimed.

7.2 Direct Compaction

At the time of an allocation failure, a process must decide whether to compact or reclaim memory. The extent of external fragmentation depends on the size of the allocation request. In our previous paper, two metrics were defined that measure external fragmentation. They are reintroduced here, but not discussed in depth except as they apply to memory compaction.

[4] As implemented by Christoph Lameter.

Fragmentation index is the first metric and is only meaningful when an allocation fails due to a lack of a suitable free area. It determines if the allocation failure was due to a lack of free memory or external fragmentation. The index is calculated as

$$F_i(j) = 1 - \frac{TotalFree/2^j}{AreasFree}$$

where $TotalFree$ is the number of free pages, j is the order of the desired allocation, and $AreasFree$ is the number of contiguous free areas of any size. When $AreasFree$ is 0, $F_i(j)$ is defined as 0.

A value tending towards 0 implies the allocation failed due to lack of memory and the process should reclaim. A value tending towards 1 implies the failure is due to external fragmentation and the process should compact. If a process tries to compact memory and fails to satisfy the allocation, it will then reclaim.

It is difficult to know in advance if a high fragmentation index is due to areas used for unmovable allocations. If it is, compacting memory will only consume CPU cycles. Hence when the index is high but before compaction starts, the index is recalculated using only blocks marked `MOVABLE`. This scan is expensive, but can take place without holding locks, and it is considerably cheaper than unnecessarily compacting memory.

When direct compaction is scanning for pages to move, only pages within `MOVABLE` areas are considered. The compaction run ends when a suitably sized area is free and the operation is considered successful. The steps taken by a process allocating a contiguous area are shown in Figure 10.

1. Attempt allocation
2. On success, return area
3. Calculate fragmentation index
4. If low memory goto 11
5. Scan areas marked MOVABLE
6. Calculate index based on MOVABLE areas alone
7. If low memory goto 11
8. Compact memory
9. Attempt allocation
10. On success, return block
11. Reclaim pages
12. Attempt allocation
13. On success, return block
14. Failed, return NULL

Figure 10: Direct Compaction

7.3 Compaction Daemon

kswapd is woken when free pages drop below a watermark to avoid processes entering direct reclaim. Similarly, **compactd** will compact memory when external fragmentation exceeds a given threshold. The daemon becomes active when woken by another process or at a timed interval.

There are two situations were **compactd** is woken up to unconditionally compact memory. When there are not enough pages free, grouping pages by mobility may be forced to mix pages of different mobility types within an area. When a non-movable allocation is improperly placed, **compactd** will be woken up. The objective is to reduce the probability that non-movable allocations will be forced to use a area reserved for `MOVABLE` because movable pages were improperly placed. When **kswapd** is woken because the high watermark for free pages is reached, **compactd** is also woken up on the assumption that movable pages can be moved from unmovable areas to the newly freed pages.

If not explicitly woken, the daemon will wake regularly and decide if compaction is necessary or not. The metric used to make this determination is called the *unusable free space index*. It measures what fraction of available free memory may be used to satisfy an allocation of a specific size. The index is calculated as

$$F_u(j) = \frac{TotalFree - \sum_{i=j}^{i=n} 2^i k_i}{TotalFree}$$

where $TotalFree$ is the number of free pages, 2^n is the largest allocation that can be satisfied, j is the order of the desired allocation, and k_i is the number of free page blocks of size 2^i. When $TotalFree$ is 0, F_u is defined as 1.

By default the daemon will only be concerned with allocations of order-3, the maximum contiguous area normally considered to be reasonable. Users of larger contiguous allocations would set this value higher. Compaction starts if the unusable free space index exceeds 0.75, implying that 75% of currently free memory is unusable for a contiguous allocation of the configured size.

In contrast to a process directly reclaiming, the migrate scanner checks all areas, not just those marked MOVABLE. The compaction daemon does not exit until the two scanners meet. The pass is considered a success if the unusable free space index was below 0.75 before the operation started, and above 0.75 after it completes.

8 Capturing Page Ranges

A significant factor in the efficiency of the reclaim algorithm is its vulnerability to racing allocators. As pressure is applied to an area of memory, pages are evicted and released. Some pages will become free very rapidly as they are clean pages; others will need expensive disk operations to record their contents. Over this period, the earliest pages are vulnerable to being used in servicing another allocation request. The loss of even a single page in the area prevents it being used for a contiguous allocation request, rendering the work Area Reclaim redundant.

8.1 Race Severity

In order to get a feel for the scale of the problem, the kernel was instrumented to record and report how often pages under Area Reclaim were reallocated to another allocator. This was measured during the **Highalloc-stress** test as described in Section 4.

The results in Figure 11 show that only a very small portion of the areas targeted for reclaim survive to be allocated. Racing allocations steal more than 97% of all areas before they coalesce. The size of the area under reclaim directly contributes to the time to reclaim and the chances of being disrupted by an allocation.

Arch	Area (MB)	Allocations		Rate (%)
		Good	Raced	
x86-64	2	500	19125	2.61
PPC64	16	152	13544	1.12

Figure 11: Racing Allocations

8.2 Capture

If the reallocation of pages being reclaimed could be prevented, there would be a significant increase in success rates and a reduction in the overall cost for releasing those areas. In order to evaluate the utility of segregating the memory being released, a prototype capture system was implemented. Benchmarking showed significant improvement in reallocation rates, but triggered unexpected interactions with overall effectiveness and increased the chances of the machine going out-of-memory. More work is required.

9 Future Considerations

The intelligent reclaim mechanism focuses on the reclaim of LRU pages because the required code is well understood and reliable. However, a significant percentage of areas are marked RECLAIMABLE, usually meaning that they are backed by the slab allocator. The slab allocator is able to reclaim pages, but like the vanilla allocator, it has no means to target which pages are reclaimed. This is particularly true when objects in the dentry cache are being reclaimed. Intelligently reclaiming slab will be harder because there may be related objects outside of the area that need to be reclaimed first. This needs investigation.

Memory compaction will linearly scan memory from the start of the address space for pages to move. This is suitable for **compactd**, but when direct compacting, it may be appropriate to select areas based on the LRU lists. Tests similar to those used when reclaiming will be used to determine if the contiguous area is likely to be successfully migrated. There are some potential issues with this, such as when the LRU pages are already at the end of memory, but it has the potential to reduce stalls incurred during compaction.

The initial results from the page capture prototype are promising but, they are not production-ready and need further development and debugging. We have considered making the direct reclaim of contiguous regions synchronous to reduce the latency and the number of pages reclaimed.

10 Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. We would like to thank Steve Fox, Tim Pepper, and Badari Pulavarty for their helpful comments and suggestions on early drafts

of this paper. A big thanks go to Dave Hansen for his insights into the interaction between grouping pages by mobility and the size of the PPC64 TLB. Finally, we would like to acknowledge the help of the community with their review and helpful commentary when developing the mechanisms described in this paper.

Legal Statement

References

[1] P. J. Denning. Virtual memory. *Computing Surveys*, 2(3):153–189, Sept. 1970.

[2] M. Gorman and A. Whitcroft. The what, the why and the where to of anti-fragmentation. In *Ottawa Linux Symposium 2006 Proceedings Volume 1*, pages 361–377, 2006.

[3] Henessny, J. L. and Patterson, D. A. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[4] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley Publishing Co., Reading, Massachusetts, 2 edition, 1973.

[5] C. Lameter. Page migration. *Kernel Source Documentation Tree*, 2006.

[6] J. E. Navarro. *Transparent operating system support for superpages*. PhD thesis, 2004. Chairman-Peter Druschel.

[7] B. Randell. A note on storage fragmentation and program segmentation. *Commun. ACM*, 12(7):365–369, 1969.

Kernel Scalability—Expanding the Horizon Beyond Fine Grain Locks

Corey Gough
Intel Corp.
corey.d.gough@intel.com

Suresh Siddha
Intel Corp.
suresh.b.siddha@intel.com

Ken Chen
Google, Inc.
kenchen@google.com

Abstract

System time is increasing on enterprise workloads as multi-core and non-uniform memory architecture (NUMA) systems become mainstream. A defining characteristic of the increase in system time is an increase in reference costs due to contention of shared resources—instances of poor memory locality also play a role. An exploration of these issues reveals further opportunities to improve kernel scalability.

This paper examines kernel lock and scalability issues encountered on enterprise workloads. These issues are examined at a software level through structure definition and organization, and at a hardware level through cache line contention characteristics and system performance metrics. Issues and opportunities are illustrated in the process scheduler, I/O paths, timers, slab, and IPC.

1 Introduction

Processor stalls due to memory latency are the most significant contributor to kernel CPI (cycles per instruction) on enterprise workloads. NUMA systems drive kernel CPI higher as cache misses that cost hundreds of clock cycles on traditional symmetric multiprocessing systems (SMP) can extend to over a thousand clock cycles on large NUMA systems. With a fixed amount of kernel instructions taking longer to execute, system time increases, resulting in fewer clock cycles for user applications. Increases in memory latency have a similar effect on the CPI of user applications. As a result, minimizing the effects of memory latency increases on NUMA systems is essential to achieving good scalability.

Cache coherent NUMA systems are designed to overcome limitations of traditional SMP systems, enabling more processors and higher bandwidth. NUMA systems split hardware resources into multiple nodes, with each node consisting of a set of one or more processors and physical memory units. Local node memory references, including references to physical memory on the same node and cache-to-cache transfers between processors on the same node, are less expensive due to lower latency. References that cross nodes, or remote node references, are done at a higher latency due to the additional costs introduced by crossing a node interconnect. As more nodes are added to a system, the cost to reference memory on a far remote node may increase further as multiple interconnects are needed to link the source and destination nodes.

In terms of memory latency, the most expensive kernel memory references can be broadly categorized as *remote memory reads*, or reads from physical memory on a different node, and as *remote cache-to-cache transfers*, or reads from a processor cache on a different node.

Many improvements have been added to the Linux kernel to reduce remote memory reads. Examples include `libnuma` and the kernel `mbind()` interface, NUMA aware slab allocation, and per-cpu scheduler group allocations. These are all used to optimize memory locality. Similarly, many improvements have been added to the kernel to improve lock sequences which decrease latency from cache-to-cache transfers. Use of the RCU (read-copy update) mechanism has enabled several scalability improvements and continues to be utilized in new development. Use of finer grain locks such as array locks for `SYSV` semaphores, conversion of the `page_table_lock` to per pmd locks for fast concurrent page faults, and per device block I/O unplug, all contribute to reduce cache line contention.

Through characterization of kernel memory latency, analysis of high latency code paths, and examination of kernel data structure layout, we illustrate further opportunities to reduce latency.

1.1 Methodology

Kernel memory latency characteristics are studied using an enterprise OLTP (online transaction processing) workload with Oracle Database 10g Release 2 on Dual-Core Intel® Itanium® 2 processors. The configuration includes a large database built on a robust, high performance storage subsystem. The workload is tuned to make best use of the kernel, utilizing fixed process priority and core affinity, optimized binding of interrupts, and use of kernel tunable settings where appropriate.

Itanium processors provide a mechanism to precisely profile cache misses. The Itanium EAR (event address registers) performance monitoring registers latch data reads that miss the L1 cache and collect an extended set of information about targeted misses. This includes memory latency in clock cycles, the address of the load instruction that caused the miss, the processor that retired the load, and the address of the data item that was missed.

Several additional processing steps are taken to supplement the data collected by hardware. Virtual data addresses are converted to a physical address during collection using the *tpa* (translate physical address) instruction and the resulting physical addresses are looked up against `SRAT` (static resource affinity table) data structures to determine node location. Kernel symbols and `gdwarf-2` annotated assembly code are analyzed to translate data addresses to kernel global variable names, structure names, and structure field names. Symbols are enhanced by inlining spinlocks and sinking them into wrapper functions, with the exported function name describing the lock being acquired, where it is acquired, and by whom it is acquired.

2 Cache Coherency

To maintain consistency between internal caches and caches on other processors, systems use a cache coherency protocol, such as the MESI protocol (modified, exclusive, shared, invalid). Each cache line contains status flags that indicate the current cache line state.

A cache line in *E* (exclusive) state indicates that the cache line does not exist in any other processor's cache. The data is clean; it matches the image in main memory.

A cache line in *S* (shared) state can exist in several caches at once. This is frequently the case for cache lines that contain data that is read, but rarely, if ever, modified.

Cache lines in *M* (modified) state are only present in one processor cache at a time. The data is dirty; it is modified and typically does not match the image in main memory. Cache lines in M state can be directly transferred to another processors cache, with the ability to satisfy another processor's read request detected through snooping. Before the cache line can be transferred, it must be written back to main memory.

Cache lines in *I* (invalid) state have been invalidated, and they cannot be transferred to another processor's cache. Cache lines are typically invalidated when there are multiple copies of the line in S state, and one of the processors needs to invalidate all copies of the line so it can modify the data. Cache lines in I state are evicted from a cache without being written back to main memory.

2.1 Cache Line Contention

Coherency is maintained at a cache line level—the coherency protocol does not distinguish between individual bytes on the same cache line. For kernel structures that fit on a single cache line, modification of a single field in the structure will result in any other copies of the cache line containing the structure to be invalidated.

Cache lines are contended when there are several threads that attempt to write to the same line concurrently or in short succession. To modify a cache line, a processor must hold it in M state. Coherency operations and state transitions are necessary to accomplish this, and these come at a cost. When a cache line containing a kernel structure is modified by many different threads, only a single image of the line will exist across the processor caches, with the cache line transferring from cache to cache as necessary. This effect is typically referred to as *cache line bouncing*.

Cache lines are also contended when global variables or fields that are frequently read are located on the same cache line as data that is frequently modified. Coherency operations and state transitions are required to transition cache lines to S state so multiple processors can hold a copy of the cache line for reading. This effect is typically referred to as *false sharing*.

Cache line contention also occurs when a thread referencing a data structure is migrated to another processor,

or when a second thread picks up computation based on a structure where a first thread left off—as is the case in interrupt handling. This behavior mimics contention between two different threads as cache lines need to be transferred from one processor's cache to another to complete the processing.

Issues with cache line contention expand further with several of the critical kernel structures spanning multiple cache lines. Even in cases where a code path is referencing only a few fields in a structure, we frequently have contention across several different cache lines.

Trends in system design further intensify issues with cache line contention. Larger caches increase the chance that a cache miss on a kernel structure will hit in a processor's cache. Doubling the number of cores and threads per processor also increases the number of threads that can concurrently reference a kernel structure. The prevalence of NUMA increases the number of nodes in a system, adding latency to the reference types mentioned in the preceding section. Cache line contention that appears fairly innocuous on small servers today has the potential to transform into significant scalability problems in the future.

3 NUMA Costs

In an experiment to characterize NUMA costs, a system is populated with two processors and 64 GB of memory using two different configurations. In the *single node* configuration, both processors and memory are placed into a single node—while the platform is NUMA, this configuration is representative of traditional SMP. In the split node configuration, processors and memory are divided evenly across two NUMA nodes. This experiment essentially toggles NUMA on and off without changing the physical computing resources, revealing interesting changes in kernel memory latency.

With the OLTP workload, scalability deteriorated when comparing the 2.6.18 kernel to the 2.6.9 kernel. On the 2.6.9 kernel, system time increases from 19% to 23% when comparing single node to split node configurations. With less processor time available for the user workload, we measure a 6% performance degradation. On the 2.6.18 kernel, comparing single node to split node configurations, system time increases from 19% to 25%, resulting in a more significant 10% performance degradation.

Kernel data cache miss latency increased 50% as we moved from single to split nodes, several times greater than the increase in user data cache miss latency. A kernel memory latency increase of this magnitude causes several of the critical kernel paths to take over 30% longer to execute—with kernel CPI increasing from 1.95 to 2.55. The two kernel paths that exhibited the largest increases in CPI were in process scheduling and I/O paths.

A comparison of single to split node configurations also reveals that a surprisingly large amount of kernel data cache misses are satisfied by cache-to-cache transfers. The most expensive cache misses were due to remote cache-to-cache transfers—reads from physical memory on a remote node accounted for a much smaller amount of the overall kernel memory latency.

4 2.6.20 Kernel

4.1 Memory Latency Characterization

Figure 1 illustrates the frequency of kernel misses at a given latency value. This data was collected using a NUMA system with two fully-populated nodes on the 2.6.20 kernel running the OLTP workload. Micro-benchmarks were used to measure latency lower bounds, confirming assumptions regarding latency of different reference types.

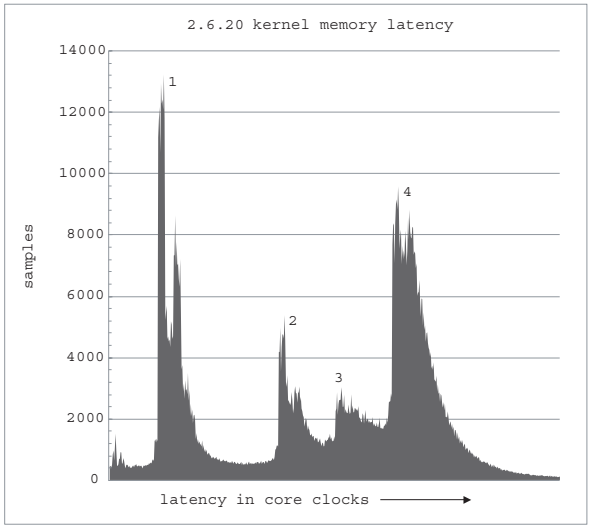


Figure 1: Kernel memory latency histogram

Four significant peaks exist on the kernel memory latency histogram. The first, largest peak corresponds to

local node cache-to-cache transfers and the second peak corresponds to local node reads from main memory. The third peak corresponds to a remote node read from main memory, and the fourth peak highlights remote node cache-to-cache transfers. Peaks do not represent absolute boundaries between different types of references as the upper bound is not fixed for any reference type. They do, however, illustrate specific latency ranges where one reference type is much more frequent than another.

46% of the kernel data cache misses are node local, however, these inexpensive misses only account for 27% of the overall kernel latency. The remote node cache misses are considerably more expensive, particularly the remote node cache-to-cache transfers. The height and width of the fourth peak captures the severity of remote node cache-to-cache transfers. The tail of the fourth peak indicates references to a heavily contended cache line, for instance a contended spinlock.

A two node configuration is used in these experiments to illustrate that latency based scalability issues are not exclusive to big iron. As we scale up the number of NUMA nodes, latencies for remote cache-to-cache transfers increase and the chance that cache-to-cache transfers will be local inexpensive references decreases.

Figure 1 also illustrates that the most frequently referenced kernel structures have a tendency to hit in processor caches. Memory reads are, relatively speaking, infrequent. Optimizations targeting the location or NUMA allocation of data do not address cache-to-cache transfers because these are infrequently read from main memory.

In examining memory latency histograms for well tuned user applications, the highest peaks are typically local node cache-to-cache transfers and local node memory reads. The most expensive reference type, the remote node cache-to-cache transfer, is typically the smallest peak. In a well tuned user application, the majority of latency comes from local node references, or is at least split evenly between local and remote node references. In comparison to user applications, the kernel's resource management, communication, and synchronization responsibilities make it much more difficult to localize processing to a single node.

4.2 Cache Line Analysis

With cache-to-cache transfers making up the two tallest peaks in the kernel memory latency histogram, and remote cache-to-cache transfers representing the most expensive overall references, 2.6.20 cache line analysis is focused on only those samples representative of a cache-to-cache transfer.

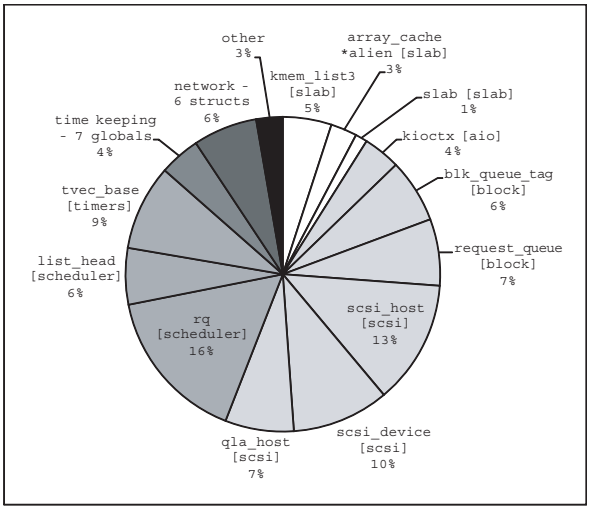


Figure 2: Top 500 kernel cache misses based on total latency

Figure 2 illustrates the top contended structures in the kernel, based on the top 500 cache lines contributing the most to kernel memory latency. While the top 500 cache lines represent only a fraction of 1% of the total kernel cache lines sampled during measurement, they account for 33% of total kernel latency. Among the thousands of structures in the kernel, only twelve structures, spread across scheduler, timers, and I/O make up the majority of the breakdown.

The chart is organized to co-locate nearby or related kernel paths. I/O includes structures from AIO, through block, through SCSI, down to the storage driver. These represent approximately half of the breakdown. Structures for the slab allocation are also included on this side of the chart as the I/O paths and related structures are the primary users of slab allocation. Scheduler paths, including the run queue, and list structures contained in the priority arrays, account for 21% of the breakdown—several percent more if we include IPC as part of this path. Dynamic timers account for 9% of the breakdown.

4.3 Cache Line Visualization

Figure 3 is an example of a concise data format used to illustrate issues with cache line contention. Every path

samples	remote	module	inst. address	inst.	function	data address	data
338	**1.18%**	**vmlinux**	**0xA000000100646036**	**cmpxchg4.acq**	**[spinlock] @ schedule**	**0xE000001000054B50**	**rq_lock**
633	**59.40%**	**vmlinux**	**0xA000000100069086**	**cmpxchg4.acq**	**[spinlock] @ try_to_wake_up**	**0xE000001000054B50**	**rq_lock**
67	22.39%	vmlinux	0xA000000100009106	ld4	ia64_spinlock_contention	0xE000001000054B50	rq_lock
9	77.78%	vmlinux	0xA000000100009126	cmpxchg4.acq	ia64_spinlock_contention	0xE000001000054B50	rq_lock
72	2.78%	vmlinux	0xA00000010006B386	cmpxchg4.acq	[spinlock] @ task_running_tick	0xE000001000054B50	rq_lock
1	0.00%	vmlinux	0xA00000010064B840	cmpxchg4.acq	[spinlock] @ wake_sleeping_dep...	0xE000001000054B50	rq_lock
3	66.67%	vmlinux	0xA00000010006BC76	ld4.acq	resched_task	0xE000001000054B50	rq_lock
18	44.44%	vmlinux	0xA000000100069EA6	cmpxchg4.acq	[spinlock] @ try_to_wake_up	0xE000001000054B50	rq_lock
5	0.00%	vmlinux	0xA000000100646130	ld8	schedule	0xE000001000054B58	nr_running
2	0.00%	vmlinux	0xA000000100646726	ld8	schedule	0xE000001000054B58	nr_running
2	0.00%	vmlinux	0xA000000100070806	ld8	try_to_wake_up	0xE000001000054B58	nr_running
1	0.00%	vmlinux	0xA00000010006CAF0	ld8	move_tasks	0xE000001000054B58	nr_running
3	0.00%	vmlinux	0xA00000010006AE60	ld8	deactivate_task	0xE000001000054B58	nr_running
1	0.00%	vmlinux	0xA0000001000753C6	ld8	scheduler_tick	0xE000001000054B58	nr_running
32	81.25%	vmlinux	0xA00000010006BBD0	ld8	__activate_task	0xE000001000054B58	nr_running
30	0.00%	vmlinux	0xA000000100068010	ld8	find_busiest_group	0xE000001000054B60	raw_weighted_load
3	0.00%	vmlinux	0xA000000100070720	ld8	try_to_wake_up	0xE000001000054B60	raw_weighted_load
2	0.00%	vmlinux	0xA000000100070740	ld8	try_to_wake_up	0xE000001000054B60	raw_weighted_load
10	0.00%	vmlinux	0xA000000100070780	ld8	try_to_wake_up	0xE000001000054B60	raw_weighted_load
1	0.00%	vmlinux	0xA00000010006CAA6	ld8	move_tasks	0xE000001000054B60	raw_weighted_load
1	0.00%	vmlinux	0xA00000010006CAF6	ld8	move_tasks	0xE000001000054B60	raw_weighted_load
6	0.00%	vmlinux	0xA000000100075486	ld8	scheduler_tick	0xE000001000054B60	raw_weighted_load
21	61.90%	vmlinux	0xA00000010006BBD6	ld8	__activate_task	0xE000001000054B60	raw_weighted_load
14	0.00%	vmlinux	0xA000000100068030	ld8	find_busiest_group	0xE000001000054B68	cpu_load[0]
2	0.00%	vmlinux	0xA000000100070786	ld8	try_to_wake_up	0xE000001000054B68	cpu_load[0]
5	0.00%	vmlinux	0xA000000100070790	ld8	try_to_wake_up	0xE000001000054B68	cpu_load[0]
2	0.00%	vmlinux	0xA000000100067F70	ld8	find_busiest_group	0xE000001000054B68	cpu_load[0]
1	0.00%	vmlinux	0xA000000100068030	ld8	find_busiest_group	0xE000001000054B70	cpu_load[1]
1	0.00%	vmlinux	0xA000000100075490	ld8	scheduler_tick	0xE000001000054B70	cpu_load[1]

Figure 3: First cache line of `rq` struct

that references a structure field satisfied by a cache-to-cache transfer is listed. Samples that are local cache hits or cache misses that reference main memory are not included, as a result, we do not see each and every field referenced in a structure. Several columns of data are included to indicate contention hotspots and to assist in locating code paths for analysis.

- `samples` – The number of sampled cache misses at a given reference point
- `remote` – The ratio of cache misses that reference a remote node compared to local node references
- `module` – The kernel module that caused the cache miss
- `instruction address` – The virtual address for the instruction that caused the miss
- `instruction` – Indicates the instruction. This illustrates the size of the field, for example ld4 is a load of a four byte field. For locks, the instruction indicates whether a field was referenced atomically, as is the case for a compare and exchange or an atomic increment / decrement
- `function` – The kernel function that caused the miss. In the case of spinlocks, we indicate the spinlock call as well as the caller function
- `data address` – The virtual address of the data item missed. This helps to illustrate spatial characteristics of contended structure fields
- `data` – Structure field name or symbolic information for the data item

4.4 Process Scheduler

The process scheduler run queue structure spans four 128 byte cache lines, with the majority of latency coming from the `try_to_wake_up()` path. Cache lines from the `rq` struct are heavily contended due to remote wakeups and local process scheduling referencing the same fields. In the first cache line of `rq`, the most expensive references to the `rq lock` and remote references from `__activate_task` come from the wakeup path.

Remote wakeups are due to two frequent paths. First, database transactions are completed when a commit occurs—this involves a write to an online log file. Hundreds of foreground processes go to sleep in the final

samples	remote	module	inst. address	inst.	function	data address	data
2	0.00%	vmlinux	0xA000000100647420	ld8	schedule	0xE000001000014B80	nr_switches
1	0.00%	vmlinux	0xA000000100646100	ld8	schedule	0xE000001000014B88	nr_uninterruptible
2	0.00%	vmlinux	0xA00000010006CB60	ld8	move_tasks	0xE000001000014B98	most_recent_timestamp
752	**63.70%**	**vmlinux**	**0xA000000100066000**	**ld8**	**idle_cpu**	**0xE000001000014BA0**	**curr (task_struct*)**
5	0.00%	vmlinux	0xA00000010006C4F0	ld8	move_tasks	0xE000001000014BA0	curr (task_struct*)
6	0.00%	vmlinux	0xA0000001006468B0	ld8	schedule	0xE000001000014BA0	curr (task_struct*)
1	0.00%	vmlinux	0xA000000100070F10	ld8	try_to_wake_up	0xE000001000014BA0	curr (task_struct*)
799	**65.96%**	**vmlinux**	**0xA000000100066006**	**ld8**	**idle_cpu**	**0xE000001000014BA8**	**idle (task_struct*)**
1	0.00%	vmlinux	0xA000000100646770	ld8	schedule	0xE000001000014BA8	idle (task_struct*)
10	0.00%	vmlinux	0xA0000001006468B6	ld8	schedule	0xE000001000014BA8	idle (task_struct*)
4	0.00%	vmlinux	0xA000000100647056	ld8	schedule	0xE000001000014BA8	idle (task_struct*)
1	0.00%	vmlinux	0xA0000001006472E0	ld8	schedule	0xE000001000014BA8	idle (task_struct*)
31	0.00%	vmlinux	0xA0000001000753A6	ld8	scheduler_tick	0xE000001000014BA8	idle (task_struct*)
4	0.00%	vmlinux	0xA000000100065C90	ld8	account_system_time	0xE000001000014BA8	idle (task_struct*)
176	**1.14%**	**vmlinux**	**0xA000000100645E46**	**ld8**	**schedule**	**0xE000001000014BA8**	**idle (task_struct*)**
2	0.00%	vmlinux	0xA000000100075526	ld8	scheduler_tick	0xE000001000014BB0	next_balance
3	0.00%	vmlinux	0xA000000100647DE6	ld8	schedule	0xE000001000014BB8	prev_mm (mm_struct*)
6	0.00%	vmlinux	0xA000000100646950	ld8	schedule	0xE000001000014BC0	active (prio_array*)
1	0.00%	vmlinux	0xA00000010006B306	ld8	task_running_tick	0xE000001000014BC0	active (prio_array*)
17	64.71%	vmlinux	0xA00000010006BB86	ld8	__activate_task	0xE000001000014BC0	active (prio_array*)
5	0.00%	vmlinux	0xA00000010006C4F6	ld8	move_tasks	0xE000001000014BC8	expired (prio_array*)
13	0.00%	vmlinux	0xA00000010006AD86	ld4	dequeue_task	0xE000001000014BD0	arrays[0].nr_active
2	50.00%	vmlinux	0xA00000010006B2B6	ld4	enqueue_task	0xE000001000014BD0	arrays[0].nr_active
1	0.00%	vmlinux	0xA00000010006AE10	ld4.acq	dequeue_task	0xE000001000014BE8	arrays[0].bitmap

Figure 4: Second cache line of `rq` struct

stage of completing a transaction and need to be woken up when their transaction is complete. Second, database processes are submitting I/O on one processor and the interrupt is arriving on another processor, causing processes waiting for I/O to be woken up on a remote processor.

A new feature being discussed in the Linux kernel community, called syslets, could be used to address issues with remote wakeups. Syslets are simple lightweight programs consisting of system calls, or *atoms*, that the kernel can execute autonomously and asynchronously. The kernel utilizes a different thread to execute the atoms asynchronously, even if a user application making the call is single threaded. Using this mechanism, user applications can wakeup foreground processes on local nodes in parallel by splitting the work between a number of syslets.

A near term approach, and one complementary to node local wakeups using syslets, would be to minimize the number of remote cache line references in the `try_to_wake_up()` path. Figure 3 confirms that the run queue is not cache line aligned. By aligning the run queue, the `rq` structure uses one less cache line. This results in a measurable performance increase as the scheduler stats `ttwu` field on the fourth cache line is brought into the third cache line alongside other data used in the wakeup.

The second cache line of the `rq` struct in figure 4 shows a high level of contention between the `idle` and `curr task_struct` pointers in `idle_task`. This issue originates from the decision to schedule a task on an idle sibling if the processor targeted for a wakeup is busy. In this path, we reference the sibling's runqueue to check if it is busy or not. When a wakeup occurs remotely, the sibling's runqueue status is also checked, resulting in additional remote cache-to-cache transfers. Load balancing at the SMT `sched_domain` happens more frequently, influencing the equal distribution of the load between siblings. Instead of checking the sibling, `idle_cpu()` can simply return the target cpu if there is more than one task running, because it's likely that siblings will also have tasks running. This change reduces contention on the second cache line, and also results in a measurable performance increase.

The third line contains one field that contributes to cache line contention, the `sd sched_domain` pointer used in `try_to_wake_up()`. The fourth `rq` cache line contains scheduler stats fields, with the most expensive remote references coming from the try to wake up count field mentioned earlier. The `list_head` structures in the top 500 are also contended in the wakeup path.

When a process is woken up, it needs to be added to the `prio_array` queue.

4.5 Timers

Processes use the per-processor `tvec_base` that corresponds to the processor they are running on when adding timers. This ensures timers are always added locally. The majority of scalability issues with timers are introduced during timer deletion. With I/O submitted on a different processor than it is completed on, it is necessary to acquire a remote `tvec_base lock` in the interrupt handler to detach a timer from that base.

Figure 5 illustrates cache line contention for the `tvec_base lock`. The `mod_timer()` path represents deletes of the block queue plug timers, which are unnecessary given we are using direct I/O in this workload. Cache line contention for the `tvec_base lock` in `del_timer()` represents deletes from the `scsi_delete_timer()` path—the result of deleting a SCSI command `eh_timeout` timer upon I/O completion.

Getting the `base_lock` for every I/O completion results in tens-of-thousands of expensive remote cache-to-cache transfers per second. The `tvec_base` locks are among the most contended locks in this workload. An alternative approach would be to batch timers for deletion, keeping track of a timer's state. A timer's state could be changed locally during an I/O completion since the timer is part of the `scsi_cmnd` struct that is read into the cache earlier in the path. Timer state could indicate both when the timer can be detached, and when the kernel can free the memory.

Where we have a large number of timers to delete across a number of `tvec_base` structures, we can prefetch locks, hiding latency for the majority of the lock references in the shadow of the first `tvec_base` lock referenced.

4.6 Slab

Submitting I/O on one processor and handling the interrupt on another processor also affects scalability of slab allocation. Comparisons between the 2.6.20 and the 2.6.9 kernel, which did not have NUMA aware slab allocation, indicate the slab allocation is hurting more than it is helping on the OLTP workload. Figures 6, 7, and 8 illustrate scalability issues introduced by freeing slabs from a remote node. The `alien` pointer in the `kmem_list3` structure and the `nodeid` in the `slab` structure are mostly read, so we may be able to reduce false sharing of these fields by moving them to another cache line. If cache lines with these fields exist in S state, several copies of the data can be held simultaneously by multiple processors, and we can eliminate some of the remote cache-to-cache transfers.

Further opportunity may exist in using `kmem_cache_alloc` calls that result in refills to free local slabs currently batched to be freed on remote processors. This has the potential to reduce creation of new slabs, localize memory references for slabs that have been evicted from the cache, and provide more efficient lock references.

A number of additional factors limit scalability. Alien array sizes are limited to `12` and are not resized based on slab tunables. In addition, the total size of alien arrays increases squarely as the number of nodes increases, putting additional memory pressure on the system.

A similar analysis for the new slub (unqueued slab) is in progress on this configuration to determine if similar issues exist.

4.7 I/O: AIO

As illustrated throughout this paper, cache line contention of I/O structures is primarily due to submitting I/O on one processor and handling the interrupt on another processor. A superior solution to this scalability issue would be to utilize hardware and device drivers that support extended message-signaled interrupts (MSI-X) and support multiple per-cpu or per-node queues. Using extended features of MSI, a device can send *messages* to a specific set of processors in the system. When an interrupt occurs, the device can decide a target processor for the interrupt. In the case of the OLTP workload, these features could be used to enable I/O submission and completion on the same processor. This approach also ensures that slab objects and timers get allocated, referenced, and freed within a local node.

Another approach, explored in earlier Linux kernels, is to delay the I/O submission so it can be executed on the processor or node which will receive the interrupt during I/O completion. Performance results with this approach

samples	remote	module	inst. address	inst.	function	data address	data
39	17.95%	vmlinux	0xA000000100009106	ld4	ia64_spinlock_contention	0xE00000100658C000	lock
9	11.11%	vmlinux	0xA000000100009126	cmpxchg4.acq	ia64_spinlock_contention	0xE00000100658C000	lock
307	**2.93%**	**vmlinux**	**0xA00000010009B4C6**	**cmpxchg4.acq**	**[spinlock] @ __mod_timer**	**0xE00000100658C000**	**lock**
861	**58.07%**	**vmlinux**	**0xA00000010009995A6**	**cmpxchg4.acq**	**[spinlock] @ del_timer**	**0xE00000100658C000**	**lock**
10	0.00%	vmlinux	0xA00000010009B666	cmpxchg4.acq	[spinlock] @ run_timer_softirq	0xE00000100658C000	lock
982	**40.22%**	**vmlinux**	**0xA0000001000996C6**	**cmpxchg4.acq**	**[spinlock] @ mod_timer**	**0xE00000100658C000**	**lock**
24	0.00%	vmlinux	0xA0000001000997E6	cmpxchg4.acq	[spinlock] @ try_to_del_timer_sync	0xE00000100658C000	lock
1	0.00%	vmlinux	0xA00000010009B966	cmpxchg4.acq	[spinlock] @ run_timer_softirq	0xE00000100658C000	lock
7	14.29%	vmlinux	0xA00000010009B466	ld8	__mod_timer	0xE00000100658C008	running_timer (timer_list*)
3	0.00%	vmlinux	0xA00000010009A906	ld8	internal_add_timer	0xE00000100658C010	timer_jiffies
78	1.28%	vmlinux	0xA00000010009B606	ld8	run_timer_softirq	0xE00000100658C010	timer_jiffies

Figure 5: Cache line with `tvec_base` struct

samples	remote	module	inst. address	inst.	function	data address	data
9	100.00%	vmlinux	0xA0000001001580F0	ld8	____cache_alloc_node	0xE00000207945BD00	slabs_partial.next (list_head*)
5	100.00%	vmlinux	0xA0000001002C3556	ld8	list_del	0xE00000207945BD00	slabs_partial.next (list_head*)
1	100.00%	vmlinux	0xA0000001002C3726	ld8	__list_add	0xE00000207945BD00	slabs_partial.next (list_head*)
2	100.00%	vmlinux	0xA0000001002C3816	ld8	list_add	0xE00000207945BD00	slabs_partial.next (list_head*)
2	100.00%	vmlinux	0xA0000001002C35C0	ld8	list_del	0xE00000207945BD08	slabs_partial.prev (list_head*)
22	59.09%	vmlinux	0xA0000001002C36B6	ld8	__list_add	0xE00000207945BD08	slabs_partial.prev (list_head*)
17	70.59%	vmlinux	0xA000000100159EE0	ld8	free_block	0xE00000207945BD08	slabs_partial.prev (list_head*)
3	66.67%	vmlinux	0xA0000001002C3556	ld8	list_del	0xE00000207945BD10	slabs_full.next (list_head*)
1	100.00%	vmlinux	0xA0000001002C3726	ld8	__list_add	0xE00000207945BD10	slabs_full.next (list_head*)
2	100.00%	vmlinux	0xA000000100158200	ld8	____cache_alloc_node	0xE00000207945BD30	free_objects
24	54.17%	vmlinux	0xA000000100159E10	ld8	free_block	0xE00000207945BD30	free_objects
408	**100.00%**	**vmlinux**	**0xA0000001001580D6**	**cmpxchg4.acq**	**[spinlock] @ ____cache_alloc_node**	**0xE00000207945BD40**	**list_lock**
17	0.00%	vmlinux	0xA00000010015A206	cmpxchg4.acq	[spinlock] @ cache_flusharray	0xE00000207945BD40	list_lock
114	**100.00%**	**vmlinux**	**0xA00000010015A3B6**	**cmpxchg4.acq**	**[spinlock] @ __drain_alien_cache**	**0xE00000207945BD40**	**list_lock**
18	0.00%	vmlinux	0xA000000100158656	cmpxchg4.acq	[spinlock] @ cache_alloc_refill	0xE00000207945BD40	list_lock
57	84.21%	vmlinux	0xA000000100009106	ld4	ia64_spinlock_contention	0xE00000207945BD40	list_lock
3	100.00%	vmlinux	0xA000000100009126	cmpxchg4.acq	ia64_spinlock_contention	0xE00000207945BD40	list_lock
727	**0.00%**	**vmlinux**	**0xA0000001001596B0**	**ld8**	**kmem_cache_free**	**0xE00000207945BD50**	**alien (array_cache**)**

Figure 6: Cache line with `kmem_list3` struct

were mixed, with a range of both good and bad results across different workloads. This model was also burdened with the complexity of handling many different corner cases.

Figure 9 illustrates the primary contention point for the `kioctx` structure, the `ctx_lock` that protects a per-process AIO context. This structure is dynamically allocated as one per process and lives throughout the lifetime of an AIO context. The first cache line is contended due to lock references in both the I/O submission and completion paths.

One approach to improve the `kioctx` structure would be to reorder structure fields to create a cache line with fields most frequently referenced in the submit path, and another cache line with fields most frequently referenced in the complete path. Further analysis is required to determine the feasibility of this concept. Contention in the I/O submission path occurs with `__aio_get_req` which needs the lock to put a iocb on a linked list and with `aio_run_iocb` which needs the lock to mark the current request as running. Contention in the I/O completion path occurs with `aio_complete` which needs the lock to put a completion event into event queue, unlink a iocb from a linked list, and perform process wakeup if there are waiters.

Raman, Hundt, and Mannarswamy introduced a technique for structure layout in multi threaded applications that optimizes for both improved spatial locality and reduces false sharing. Reordering fields in a few optimized structures in the HP-UX operating system kernel improved performance up to 3.2% on enterprise workloads. Structures across the I/O layers appear to be good candidates for such optimization.

The `aio_complete` function is a heavyweight function with very long lock hold times. Having a lock hold time dominated by one processor contributes to longer contention wait time with other shorter paths occurring frequently on all the other processors. In some cases, increasing the number of AIO contexts may help address this.

Figure 10 illustrates the AIO context internal structure used to track the kernel mapping of AIO `ring_info`

samples	remote	module	inst. address	inst.	function	data address	data
9	0.00%	vmlinux	0xA000000100159740	ld4	kmem_cache_free	0xE000000128DA5580	avail
1	0.00%	vmlinux	0xA000000100159770	ld4	kmem_cache_free	0xE000000128DA5580	avail
5	0.00%	vmlinux	0xA00000010015A3F0	ld4	__drain_alien_cache	0xE000000128DA5584	limit
8	0.00%	vmlinux	0xA000000100159746	ld4	kmem_cache_free	0xE000000128DA5584	limit
30	0.00%	vmlinux	0xA000000100009106	ld4	ia64_spinlock_contention	0xE000000128DA5590	lock
2	0.00%	vmlinux	0xA000000100009126	cmpxchg4.acq	ia64_spinlock_contention	0xE000000128DA5590	lock
1793	**0.00%**	**vmlinux**	**0xA000000100159716**	**cmpxchg4.acq**	**[spinlock] @ cache_free_alien**	**0xE000000128DA5590**	**lock**
2	0.00%	vmlinux	0xA0000001002C07A0	ld8	__copy_user	0xE000000128DA5598	objpp (entry[0]*)
2	0.00%	vmlinux	0xA0000001002C07A6	ld8	__copy_user	0xE000000128DA55A0	objpp[i]
3	0.00%	vmlinux	0xA000000100159C20	ld8	free_block	0xE000000128DA55A0	objpp[i]
4	0.00%	vmlinux	0xA000000100159C20	ld8	free_block	0xE000000128DA55B0	objpp[i]
1	0.00%	vmlinux	0xA000000100159C20	ld8	free_block	0xE000000128DA55C0	objpp[i]
1	0.00%	vmlinux	0xA000000100159C20	ld8	free_block	0xE000000128DA55D8	objpp[i]
2	0.00%	vmlinux	0xA000000100159C20	ld8	free_block	0xE000000128DA55E0	objpp[i]

Figure 7: Cache line with alien `array_cache` struct

samples	remote	module	inst. address	inst.	function	data address	data
4	100.00%	vmlinux	0xA0000001002C3556	ld8	list_del	0xE0000010067D0000	list.next (list_head*)
1	100.00%	vmlinux	0xA0000001002C35B0	ld8	list_del	0xE0000010067D0000	list.next (list_head*)
1	100.00%	vmlinux	0xA0000001002C3726	ld8	__list_add	0xE0000010067D0000	list.next (list_head*)
88	98.86%	vmlinux	0xA0000001002C35C0	ld8	list_del	0xE0000010067D0008	list.prev (list_head*)
79	100.00%	vmlinux	0xA0000001002C36B6	ld8	__list_add	0xE0000010067D0008	list.prev (list_head*)
64	100.00%	vmlinux	0xA000000100158160	ld4	____cache_alloc_node	0xE0000010067D0020	inuse
2	100.00%	vmlinux	0xA000000100159DF6	ld4	free_block	0xE0000010067D0020	inuse
4	100.00%	vmlinux	0xA000000100158226	ld4	____cache_alloc_node	0xE0000010067D0024	free (kmem_bufctl_t)
513	**45.03%**	**vmlinux**	**0xA000000100159636**	**ld2**	**kmem_cache_free**	**0xE0000010067D0028**	**nodeid**

Figure 8: Cache line with `slab` struct

and the AIO event buffer. A small percentage of the cache line contention comes from with I/O submit path where kernel needs to look up a kernel mapping for an AIO `ring_info` structure. A large percentage of the cache line contention comes from the I/O interrupt path, where `aio_complete` needs to lookup the kernel mapping of the AIO event buffer. In this case, the majority of contention comes from many I/O completions happening one after the other.

4.8 I/O: Block

Figure 11 illustrates contention over `request_queue` structures. The primary contention points are between the `queue_flags` used in the submit path to check block device queue's status and the `queue_lock` in the return path to perform a reference count on `device_busy`. This suggests further opportunity for structure ordering based on submit and complete paths.

Figure 12 illustrates the `blk_queue_tag` structure. Lack of alignment with this structure causes unnecessary cache line contention as the size of `blk_queue_tag` is less than half a cache line, so we end up with portions of two to three different tags sharing the same 128 byte cache line. Cache lines are frequently bounced between processors with multiple tags from independent devices sharing the same cache line.

4.9 I/O: SCSI

Both the `scsi_host` and `scsi_device` structures span several cache lines and may benefit from reordering. Both structures feature a single cache line with multiple contended fields, and several other lines which have a single field that is heavily contended.

Figure 13 illustrates the most heavily contended cache line of the `scsi_host` structure. The primary source of contention is in the submit path with an unconditional spin lock acquire in `__scsi_put_command` to put a `scsi_cmnd` on a local `free_list`, and reads of the `cmd_pool` field. Multiple locks on the same cache line is detrimental as it slows progress in the submit path

samples	remote	module	inst. address	inst.	function	data address	data
1	0.00%	vmlinux	0xA0000001001AB240	fetchadd4.rel	lookup_ioctx	0xE0000010252A1900	users
2	0.00%	vmlinux	0xA0000001001ABD60	fetchadd4.rel	sys_io_getevents	0xE0000010252A1900	users
44	2.27%	vmlinux	0xA0000001001AB1A6	ld8	lookup_ioctx	0xE0000010252A1910	mm (mm_struct*)
14	71.43%	vmlinux	0xA000000100066E56	cmpxchg4.acq	[spinlock] @ wake_up_common	0xE0000010252A1920	wait.lock (wait_queue_head)
1	0.00%	vmlinux	0xA0000001000009126	cmpxchg4.acq	ia64_spinlock_contention	0xE0000010252A1920	wait.lock (wait_queue_head)
4	0.00%	vmlinux	0xA0000001000B9F76	cmpxchg4.acq	[spinlock] @ add_wait_queue_exc..	0xE0000010252A1920	wait.lock (wait_queue_head)
6	0.00%	vmlinux	0xA0000001000B9FD6	cmpxchg4.acq	[spinlock] @ remove_wait_queue	0xE0000010252A1920	wait.lock (wait_queue_head)
1	0.00%	vmlinux	0xA0000001002C3556	ld8	list_del	0xE0000010252A1928	wait.next
5	60.00%	vmlinux	0xA000000100065EB0	ld8	__wake_up_common	0xE0000010252A1928	wait.next
4	75.00%	vmlinux	0xA0000001001A9F90	ld8	aio_complete	0xE0000010252A1928	wait.next
372	**1.34%**	**vmlinux**	**0xA0000001001AA2D6**	**cmpxchg4.acq**	**[spinlock] @ aio_run_iocb**	**0xE0000010252A1938**	**ctx_lock**
1066	**1.13%**	**vmlinux**	**0xA0000001001AAA36**	**cmpxchg4.acq**	**[spinlock] @ __aio_get_req**	**0xE0000010252A1938**	**ctx_lock**
14	0.00%	vmlinux	0xA0000001001ACC36	cmpxchg4.acq	[spinlock] @ io_submit_one	0xE0000010252A1938	ctx_lock
1214	**62.03%**	**vmlinux**	**0xA0000001001A8FB6**	**cmpxchg4.acq**	**[spinlock] @ aio_complete**	**0xE0000010252A1938**	**ctx_lock**
181	**34.25%**	**vmlinux**	**0xA000000100009106**	**ld4**	**ia64_spinlock_contention**	**0xE0000010252A1938**	**ctx_lock**
23	26.09%	vmlinux	0xA000000100009126	cmpxchg4.acq	ia64_spinlock_contention	0xE0000010252A1938	ctx_lock
33	69.70%	vmlinux	0xA0000001001A9586	ld4.acq	__aio_put_req	0xE0000010252A1938	ctx_lock
39	82.05%	vmlinux	0xA0000001001A97D6	ld4.acq	__aio_put_req	0xE0000010252A1938	ctx_lock
52	0.00%	vmlinux	0xA0000001001A99B6	cmpxchg4.acq	[spinlock] @ aio_put_req	0xE0000010252A1938	ctx_lock
4	0.00%	vmlinux	0xA0000001001AABD0	ld4	__aio_get_req	0xE0000010252A193C	reqs_active
28	64.29%	vmlinux	0xA0000001001A98C6	ld4	__aio_put_req	0xE0000010252A193C	reqs_active
2	50.00%	vmlinux	0xA0000001002C35C0	ld8	list_del	0xE0000010252A1948	active_reqs.prev
24	4.17%	vmlinux	0xA0000001001A80D0	ld8	aio_read_evt	0xE0000010252A1978	ring_info.ring_pages (page**)
7	0.00%	vmlinux	0xA0000001001A82B0	ld8	aio_read_evt	0xE0000010252A1978	ring_info.ring_pages (page**)
1	0.00%	vmlinux	0xA0000001001AAA96	ld8	__aio_get_req	0xE0000010252A1978	ring_info.ring_pages (page**)
3	33.33%	vmlinux	0xA0000001001A9C56	ld8	aio_complete	0xE0000010252A1978	ring_info.ring_pages (page**)
16	75.00%	vmlinux	0xA0000001001A9D90	ld8	aio_complete	0xE0000010252A1978	ring_info.ring_pages (page**)

Figure 9: Cache line with `kioctx` struct

samples	remote	module	inst. address	inst.	function	data address	data
55	1.82%	vmlinux	0xA0000001001A8216	cmpxchg4.acq	[spinlock] @ aio_read_evt	0xE0000010252A1980	lock
2	0.00%	vmlinux	0xA0000001001A8240	ld4	aio_read_evt	0xE0000010252A1990	nr
2	0.00%	vmlinux	0xA0000001001A83E0	ld4	aio_read_evt	0xE0000010252A1990	nr
3	0.00%	vmlinux	0xA0000001001A9CD0	ld4	aio_complete	0xE0000010252A1994	tail
41	0.00%	vmlinux	0xA0000001001A80E0	ld8	aio_read_evt	0xE0000010252A1998	internal_pages[0] (page*)
415	**1.20%**	**vmlinux**	**0xA0000001001AAAA6**	**ld8**	**__aio_get_req**	**0xE0000010252A1998**	**internal_pages[0] (page*)**
1033	**58.95%**	**vmlinux**	**0xA0000001001A9C76**	**ld8**	**aio_complete**	**0xE0000010252A1998**	**internal_pages[0] (page*)**
1	100.00%	vmlinux	0xA0000001001A9DB0	ld8	aio_complete	0xE0000010252A1998	internal_pages[0] (page*)
1	100.00%	vmlinux	0xA0000001001A9DB0	ld8	aio_complete	0xE0000010252A19A0	internal_pages[1] (page*)
3	0.00%	vmlinux	0xA0000001001A82D0	ld8	aio_read_evt	0xE0000010252A19A8	internal_pages[2] (page*)

Figure 10: Cache line with `ring_info` struct

with the cache line bouncing between processors as they submit I/O.

Figure 14 illustrates contention in the `scsi_device` struct with `scsi_request_fn()` referencing the host pointer to process the I/O, and the low level driver checking the `scsi_device queue_depth` to determine whether it should change the `queue_depth` on a specific SCSI device. These reads lead to false sharing on the `queue_depth` field, as `scsi_adjust_queue_depth()` is not called during the workload. In this case, the `scsi_qla_host flags` could be extended to indicate whether `queue_depth` needs to be adjusted, resulting in the interrupt service routine making frequent local references rather than expensive remote references.

Figure 15 illustrates further contention between the I/O submit and complete paths as `sd_init_command()` reads of the `timeout` and `changed` fields in the submit path conflict with writes of `iodone_cnt` in the complete path.

5 Conclusions

Kernel memory latency increases significantly as we move from traditional SMP to NUMA systems, resulting in less processor time available for user workloads. The most expensive references, remote cache-to-cache transfers, primarily come from references to a select few structures in the process wakeup and I/O paths. Several approaches may provide mechanisms to alleviate

samples	remote	module	inst. address	inst.	function	data address	data
3	66.67%	vmlinux	0xA0000001001D1F26	ld8	__blockdev_direct_IO	0xE00000012C905C00	backing_dev_info.unplug_io_fn
1	100.00%	vmlinux	0xA00000010029A0A0	ld8	blk_backing_dev_unplug	0xE00000012C905C08	backing_dev_info.unplug_io_data
1	0.00%	scsi	0xA00000021E2A9B80	ld8	scsi_run_queue	0xE00000012C905C10	queuedata
147	**59.86%**	**vmlinux**	**0xA0000001002942E6**	**ld4.acq**	**generic_make_request**	**0xE00000012C905C28**	**queue_flags**
1	0.00%	vmlinux	0xA000000100294750	ld4.acq	__freed_request	0xE00000012C905C28	queue_flags
3	33.33%	vmlinux	0xA000000100294766	cmpxchg4.acq	__freed_request	0xE00000012C905C28	queue_flags
1	0.00%	vmlinux	0xA000000100294780	ld4.acq	__freed_request	0xE00000012C905C28	queue_flags
6	16.67%	vmlinux	0xA000000100294796	cmpxchg4.acq	__freed_request	0xE00000012C905C28	queue_flags
20	55.00%	vmlinux	0xA00000010296B66	cmpxchg4.acq	blk_remove_plug	0xE00000012C905C28	queue_flags
1	100.00%	scsi	0xA00000021E2AF2D6	cmpxchg4.acq	scsi_request_fn	0xE00000012C905C28	queue_flags
1	0.00%	vmlinux	0xA000000100297456	ld4.acq	blk_plug_device	0xE00000012C905C28	queue_flags
4	25.00%	vmlinux	0xA000000100297486	cmpxchg4.acq	blk_plug_device	0xE00000012C905C28	queue_flags
3	66.67%	vmlinux	0xA000000100295930	ld4.acq	get_request	0xE00000012C905C28	queue_flags
6	50.00%	vmlinux	0xA00000010028DE96	ld4.acq	elv_insert	0xE00000012C905C28	queue_flags
6	50.00%	vmlinux	0xA00000010029C046	cmpxchg4.acq	[spinlock] @ __make_request	0xE00000012C905C30	__queue_lock
3	66.67%	vmlinux	0xA000000100290156	cmpxchg4.acq	[spinlock] @ blk_run_queue	0xE00000012C905C30	__queue_lock
18	50.00%	vmlinux	0xA00000010029AB56	cmpxchg4.acq	[spinlock] @ generic_unplug_d..	0xE00000012C905C30	__queue_lock
5	40.00%	scsi	0xA00000021E2ACF06	cmpxchg4.acq	[spinlock] @ scsi_device_unbusy	0xE00000012C905C30	__queue_lock
28	28.57%	vmlinux	0xA000000100009106	ld4	ia64_spinlock_contention	0xE00000012C905C30	__queue_lock
13	23.08%	vmlinux	0xA00000010029B2B6	cmpxchg4.acq	[spinlock] @ __make_request	0xE00000012C905C30	__queue_lock
16	56.25%	scsi	0xA00000021E2A9336	cmpxchg4.acq	[spinlock] @ scsi_end_request	0xE00000012C905C30	__queue_lock
5	20.00%	scsi	0xA00000021E2AF806	cmpxchg4.acq	[spinlock] @ scsi_request_fn	0xE00000012C905C30	__queue_lock
3	33.33%	scsi	0xA00000021E2AFA36	cmpxchg4.acq	[spinlock] @ scsi_request_fn	0xE00000012C905C30	__queue_lock
1	0.00%	vmlinux	0xA00000010029C000	ld8	__make_request	0xE00000012C905C38	queue_lock*
124	**33.06%**	**scsi**	**0xA00000021E2ACEE6**	**ld8**	**scsi_device_unbusy**	**0xE00000012C905C38**	**queue_lock***
2	50.00%	vmlinux	0xA000000100298EE0	ld8	blk_run_queue	0xE00000012C905C38	queue_lock*
5	60.00%	vmlinux	0xA000000100299006	ld8	blk_run_queue	0xE00000012C905C38	queue_lock*
10	30.00%	scsi	0xA00000021E2AB540	ld8	scsi_end_request	0xE00000012C905C38	queue_lock*
1	0.00%	scsi	0xA00000021E2AB5C6	ld8	scsi_end_request	0xE00000012C905C38	queue_lock*
11	45.45%	scsi	0xA00000021E2AF7C6	ld8	scsi_request_fn	0xE00000012C905C38	queue_lock*
1	100.00%	scsi	0xA00000021E2AF9B6	ld8	scsi_request_fn	0xE00000012C905C38	queue_lock*
1	0.00%	scsi	0xA00000021E2AF9F6	ld8	scsi_request_fn	0xE00000012C905C38	queue_lock*

Figure 11: Cache line with `request_queue` struct

samples	remote	module	inst. address	inst.	function	data address	data
119	**56.30%**	**vmlinux**	**0xA000000100299F06**	**ld8**	**blk_queue_start_tag**	**0xE000000128E8BC08**	**tag_map**
128	**54.69%**	**vmlinux**	**0xA000000100299F00**	**ld4**	**blk_queue_start_tag**	**0xE000000128E8BC24**	**max_depth**
113	**28.32%**	**vmlinux**	**0xA000000100291C26**	**ld4**	**blk_queue_end_tag**	**0xE000000128E8BC28**	**real_max_depth**
122	**53.28%**	**vmlinux**	**0xA0000001002B30E0**	**ld8**	**find_next_zero_bit**	**0xE000000128E8BC40**	**tag_map**
96	12.50%	vmlinux	0xA000000100291C66	ld4	blk_queue_end_tag	0xE000000128E8BC40	tag_map
16	50.00%	vmlinux	0xA000000100299F76	cmpxchg4.acq	blk_queue_start_tag	0xE000000128E8BC40	tag_map

Figure 12: Cache line with `blk_queue_tag` struct

scalability issues in the future. Examples include hardware and software utilizing message-signaled interrupts (MSI-X) with per-cpu or per-node queues, and syslets. In the near term, several complementary approaches can be taken to improve scalability.

Improved code sequences reducing remote cache-to-cache transfers, similar to the optimization targeting `idle_cpu()` calls or reducing the number of `rq` cache lines referenced in a remote wakeup are beneficial and need to be pursued further. Proper alignment of structures also reduces cache-to-cache transfers, as illustrated in the `blk_queue_tag` structure. Opportunities exist in ordering structure fields to increase cache line sharing and reduce contention. Examples include separating read only data from contended read / write fields, and careful placement or isolation of frequently referenced spinlocks. In the case of `scsi_host` and to a lesser extent `kioctx`, several atomic semaphores placed on the same cache quickly increase contention for a structure.

Data ordering also extends to good use of `__read_mostly` attributes for global variables, which also have a significant impact. In the 2.6.9 kernel, false sharing occurred between `inode_lock` and a hot read-only global on the same cache line, ranking it number 1 in the top 50. In the 2.6.20 kernel, the hot read-only global was given the `__read_mostly` attribute, and the cache line with `inode_lock` dropped out of the top 50 cache lines. The combination of the two new cache lines contributed 45% less latency with the globals separated.

Operations that require a lock to be held for a single frequently-occurring operation, such a timer delete from

samples	remote	module	inst. address	inst.	function	data address	data
852	**50.23%**	**scsi**	**0xA00000021E29DA80**	**ld8**	**__scsi_get_command**	**0xE0000010260C8020**	**cmd_pool**
2	100.00%	scsi	0xA00000021E29DF10	ld8	__scsi_put_command	0xE0000010260C8020	cmd_pool
292	**69.86%**	**scsi**	**0xA00000021E29C536**	**cmpxchg4.acq**	**[spinlock] @ scsi_put_command**	**0xE0000010260C8028**	**free_list_lock**
3	0.00%	scsi	0xA00000021E29DE60	ld8	__scsi_put_command	0xE0000010260C8030	free_list.next
5	60.00%	scsi	0xA00000021E2AA0F6	ld8	scsi_run_queue	0xE0000010260C8040	starved_list.next
292	**40.75%**	**qla**	**0xA00000021E4063E6**	**cmpxchg4.acq**	**[spinlock] @ qla2x00_queuecom..**	**0xE0000010260C8050**	**default_lock**
41	58.54%	scsi	0xA00000021E29C416	cmpxchg4.acq	[spinlock] @ scsi_dispatch_cmd	0xE0000010260C8050	default_lock
98	50.00%	vmlinux	0xA000000100009106	ld4	ia64_spinlock_contention	0xE0000010260C8050	default_lock
14	50.00%	vmlinux	0xA000000100009126	cmpxchg4.acq	ia64_spinlock_contention	0xE0000010260C8050	default_lock
95	49.47%	scsi	0xA00000021E2AF456	cmpxchg4.acq	[spinlock] @ scsi_request_fn	0xE0000010260C8050	default_lock
97	67.01%	scsi	0xA00000021E2A95B6	cmpxchg4.acq	[spinlock] @ scsi_device_unbu..	0xE0000010260C8050	default_lock
42	78.57%	scsi	0xA00000021E2A9736	cmpxchg4.acq	[spinlock] @ scsi_run_queue	0xE0000010260C8050	default_lock
27	37.04%	qla	0xA00000021E4062B6	ld8	qla2x00_queuecommand	0xE0000010260C8058	host_lock*
197	**57.36%**	**qla**	**0xA00000021E4063A6**	**ld8**	**qla2x00_queuecommand**	**0xE0000010260C8058**	**host_lock***
41	48.78%	scsi	0xA00000021E29E3A0	ld8	scsi_dispatch_cmd	0xE0000010260C8058	host_lock*
3	66.67%	scsi	0xA00000021E29E526	ld8	scsi_dispatch_cmd	0xE0000010260C8058	host_lock*
355	**65.63%**	**scsi**	**0xA00000021E2ACDB0**	**ld8**	**scsi_device_unbusy**	**0xE0000010260C8058**	**host_lock***
14	50.00%	scsi	0xA00000021E2ACEA6	ld8	scsi_device_unbusy	0xE0000010260C8058	host_lock*
123	**55.28%**	**scsi**	**0xA00000021E2AF436**	**ld8**	**scsi_request_fn**	**0xE0000010260C8058**	**host_lock***
34	50.00%	scsi	0xA00000021E2AF746	ld8	scsi_request_fn	0xE0000010260C8058	host_lock*
38	76.32%	scsi	0xA00000021E2A9E80	ld8	scsi_run_queue	0xE0000010260C8058	host_lock*

Figure 13: Cache line with `scsi_host` struct

samples	remote	module	inst. address	inst.	function	data address	data
26	38.46%	scsi	0xA00000021E29EB10	ld8	scsi_put_command	0xE00000012464C000	host (scsi_host*)
4	25.00%	scsi	0xA00000021E29ED10	ld8	scsi_get_command	0xE00000012464C000	host (scsi_host*)
11	45.45%	scsi	0xA00000021E2ACD90	ld8	scsi_device_unbusy	0xE00000012464C000	host (scsi_host*)
22	59.09%	scsi	0xA00000021E29CDE0	ld8	scsi_finish_command	0xE00000012464C000	host (scsi_host*)
90	**54.44%**	**scsi**	**0xA00000021E2AF0F0**	**ld8**	**scsi_request_fn**	**0xE00000012464C000**	**host (scsi_host*)**
1	100.00%	scsi	0xA00000021E2A9B96	ld8	scsi_run_queue	0xE00000012464C000	host (scsi_host*)
2	100.00%	scsi	0xA00000021E29DFB0	ld8	scsi_dispatch_cmd	0xE00000012464C000	host (scsi_host*)
2	0.00%	scsi	0xA00000021E2AAF10	ld8	scsi_next_command	0xE00000012464C008	request_queue (request_queue*)
1	0.00%	scsi	0xA00000021E2ACF26	ld8	scsi_device_unbusy	0xE00000012464C008	request_queue (request_queue*)
1	100.00%	scsi	0xA00000021E2AB826	ld8	scsi_io_completion	0xE00000012464C008	request_queue (request_queue*)
6	33.33%	scsi	0xA00000021E29C2F6	cmpxchg4.acq	[spinlock] @ scsi_put_command	0xE00000012464C034	list_lock
21	52.38%	scsi	0xA00000021E29C3B6	cmpxchg4.acq	[spinlock] @ scsi_get_command	0xE00000012464C034	list_lock
2	0.00%	scsi	0xA00000021E2AF686	ld8	scsi_request_fn	0xE00000012464C048	starved_entry.next
92	**35.87%**	**qla**	**0xA00000021E420D60**	**ld4**	**qla2x00_process_completed_req...**	**0xE00000012464C060**	**queue_depth**
2	100.00%	qla	0xA00000021E41EDF0	ld4	qla2x00_start_scsi	0xE00000012464C074	lun

Figure 14: First cache line of `scsi_device` struct

a `tvec_base`, can be batched so many operations can be completed with a single lock reference. In addition, iterating across all nodes for per-node operations may provide an opportunity to prefetch locks and data ahead for the next node to be processed, hiding the latency of expensive memory references. This technique may be utilized to improve freeing of slab objects.

Through a combination of improvements to the existing kernel sources, new feature development targeting the reduction of remote cache-to-cache transfers, and improvements in hardware and software capabilities to identify and characterize cache line contention, we can ensure improved scalability on future platforms.

6 Acknowledgements

Special thanks to Chinang Ma, Doug Nelson, Hubert Nueckel, and Peter Wang for measurement and analysis contributions to this paper. We would also like to thank Collin Terrell for draft reviews.

References

[1] *MSI-X ECN Against PCI Conventional 2.3*. `http://www.pcisig.com/specifications/conventional/`.

[2] Intel Corporation. Intel vtune performance analyzer for linux – *dsep and sfdump5*. `http://www.intel.com/cd/software/`

samples	remote	module	inst. address	inst.	function	data address	data
1	100.00%	sd	0xA00000021E247D80	ld4	sd_init_command	0xE00000012B304880	sector_size
1	0.00%	qla	0xA00000021E406100	ld8	qla2x00_queuecommand	0xE00000012B304888	hostdata (void*)
67	50.75%	sd	0xA00000021E2476E6	ld8	sd_init_command	0xE00000012B3048C8	changed
7	71.43%	scsi	0xA00000021E2A9B90	ld8	scsi_run_queue	0xE00000012B3048C8	changed
13	46.15%	scsi	0xA00000021E29E260	fetchadd4.rel	scsi_dispatch_cmd	0xE00000012B3048D8	iorequest_cnt
104	**27.88%**	**scsi**	**0xA00000021E29D2B0**	**fetchadd4.rel**	**__scsi_done**	**0xE00000012B3048DC**	**iodone_cnt**
129	**48.06%**	**sd**	**0xA00000021E247566**	**ld4**	**sd_init_command**	**0xE00000012B3048E4**	**timeout**

Figure 15: Second cache line of `scsi_device` struct

products/asmo-na/eng/vtune/239144.htm.

[3] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manuals*. http://developer.intel.com/products/processor/manuals/index.htm.

[4] Intel Corporation. *Intel Itanium 2 Architecture Software Developer's Manuals*. http://developer.intel.com/design/itanium2/documentation.htm.

[5] Hewlett-Packard Laboratories. q-tools project. http://www.hpl.hp.com/research/linux/q-tools/.

[6] Ingo Molnar. *Syslets, generic asynchronous system call support*. http://redhat.com/~mingo/syslet-patches/.

[7] Easwaran Raman, Robert Hundt, and Sandya Mannarswamy. *Structure Layout Optimization for Multithreaded Programs*. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, 2007.

7 Legal

Kdump: Smarter, Easier, Trustier

Vivek Goyal
IBM
vgoyal@in.ibm.com

Neil Horman
Red Hat, Inc.
nhorman@redhat.com

Ken'ichi Ohmichi
NEC Soft
oomichi@mxs.nes.nec.co.jp

Maneesh Soni
IBM
maneesh@in.ibm.com

Ankita Garg
IBM
ankita@in.ibm.com

Abstract

Kdump, a kexec based kernel crash dumping mechanism, has witnessed a lot of new significant development activities recently. Features like, the relocatable kernel, dump filtering, and initrd based dumping have enhanced kdump capabilities are important steps towards making it a more reliable and easy to use solution. New tools like Linux Kernel Dump Test Module (LKDTM) provide an opportunity to automate kdump testing. It can be especially useful for distributions to detect and weed out regressions relatively quickly. This paper presents various newly introduced features and provides implementation details wherever appropriate. It also briefly discusses the future directions, such as early boot crash dumping.

1 Introduction

Kdump is a kernel crash dumping mechanism where a pre-loaded kernel is booted in, to capture the crash dump after a system crash [12]. This pre-loaded kernel, often called as capture kernel, runs from a different physical memory area than the production kernel or regular kernel. As of today, a capture kernel is specifically compiled and linked for a specific memory location, and is shipped as an extra kernel to capture the dump. A relocatable kernel implementation gets rid of the requirement to run the kernel from the address it has been compiled for, instead one can load the kernel at a different address and run it from there. Effectively the distributions and kdump users don't have to build an extra kernel to capture the dump, enhancing ease of use. Section 2 provides the details of relocatable kernel implementation.

Modern machines are being shipped with bigger and bigger RAMs and a high end configuration can possess a tera-byte of RAM. Capturing the contents of the entire RAM would result in a proportionately large core file and managing a tera-byte file can be difficult. One does not need the contents of entire RAM to debug a kernel problem and many pages like userspace pages can be filtered out. Now an open source userspace utility is available for dump filtering and Section 3 discusses the working and internals of the utility.

Currently, a kernel crash dump is captured with the help of init scripts in the userspace in the capture kernel. This approach has some drawbacks. Firstly, it assumes that the root filesystem did not get corrupted and is still mountable in the second kernel. Secondly, minimal work should be done in second kernel and one need not have to run various user space init scripts. This led to the idea of building a custom initial ram disk (initrd) to capture the dump and improve the reliability of the operation. Various implementation details of initrd based dumping are presented in Section 4.

Section 5 discusses the Linux Kernel Dump Test Module (LKDTM), a kernel module, which allows one to set up and trigger crashes from various kernel code paths at run time. It can be used to automate kdump testing procedure to identify bugs and eliminate regressions with lesser efforts. This paper also gives a brief update on device driver hardening efforts in Section 6 and concludes with future work in Section 7.

2 Relocatable bzImage

Generally, the Linux® kernel is compiled for a fixed address and it runs from that address. Traditionally, for i386 and x86_64 architectures, it has been compiled and run from 1MB physical memory location. Later, Eric W. Biederman introduced a config option,

CONFIG_PHYSICAL_START, which allowed a kernel to be compiled for a different address. This option effectively shifted the kernel in virtual address space and one could compile and run the kernel from a physical address say, 16MB. Kdump used this feature and built a separate kernel for capturing the dump. This kernel was specifically built to run from a reserved memory location.

The requirement of building an extra kernel for dump capturing has not gone over too well with distributions, as they end up shipping an extra kernel binary. Apart from disk space requirements, it also led to increased efforts in terms of supporting and testing this extra kernel. Also from a user's perspective, building an extra kernel is cumbersome.

The solution to the above problem is a relocatable kernel, where the same kernel binary can be loaded and run from a different address than what it has been compiled for. Jan Kratochvil had posted a set of prototype patches to kick- off the discussion on the fastboot mailing list [6]. Later, Eric W. Biederman came up with another set of patches and posted them to LKML for review [8]. Finally, Vivek Goyal picked up Eric's patches, cleaned those up, fixed a number of bugs, incorporated various review comments, and went through multiple rounds of reposting to LKML for inclusion into the mainline kernel. Relocatable kernel implementation is very architecture dependent and support for i386 architecture has been merged with version 2.6.20 of the mainline kernel. Patches for x86_64 have been posted on LKML [11] and are now part of -mm. Hopefully, these will be merged with mainline kernel soon.

2.1 Design Approaches

The following design approaches have been discussed for the relocatable bzImage implementation.

- **Modify kernel text/data mapping at run time**
  At run time, the kernel determines where it has been loaded by the boot-loader and it updates its page tables to reflect the right mapping between kernel virtual and physical addresses for kernel text and data. This approach has been adopted for the x86_64 implementation.
- **Relocate using relocation information**
  This approach forces the linker to generate relocation information. These relocations are processed and packed into the bzImage. The uncompressed kernel code decompresses the kernel, performs the relocations, and transfers control to the protected mode kernel. This approach has been adopted by the i386 implementation.

2.2 Design Details (i386)

In i386, kernel text and data are part of linearly mapped region which has got hard-coded assumptions about virtual to physical address mapping. Hence, it is probably difficult to adopt the modifying the page table approach for implementing a relocatable kernel. Instead, a simpler, non-intrusive approach is to ask the linker to generate relocation information, pack this relocation information into bzImage, and the uncompressed kernel code can process these relocations before jumping to the 32-bit kernel entry point (`startup_32()`).

2.2.1 Relocation Information Generation

Relocation information can be generated in many ways. The initial experiment was to compile the kernel as shared object file (`-shared`) which generated the relocation entries. Eric had posted the patches for this approach [9] but it was found that the linker also generated the relocation entries for absolute symbols (for some historical reason) [1]. By definition, absolute symbols are not to be relocated, but, with this approach, absolute symbols also ended up being relocated. Hence this method did not prove to be a viable one.

Later, a different approach was taken where the i386 kernel is built with the linker option `--emit-relocs`. This option builds an executable vmlinux and still retains relocation information in a fully linked executable. This increases the size of vmlinux by around 10%, though this information is discarded at runtime. The kernel build process goes through these relocation entries and filters out PC relative relocations, as these don't have to be adjusted if bzImage is loaded at a different physical address. It also filters out the relocations generated with respect to absolute symbols because absolute symbols don't have to be relocated. The rest of the relocation offsets are packed into the compressed vmlinux. Figure 1 depicts the new i386 bzImage build process.

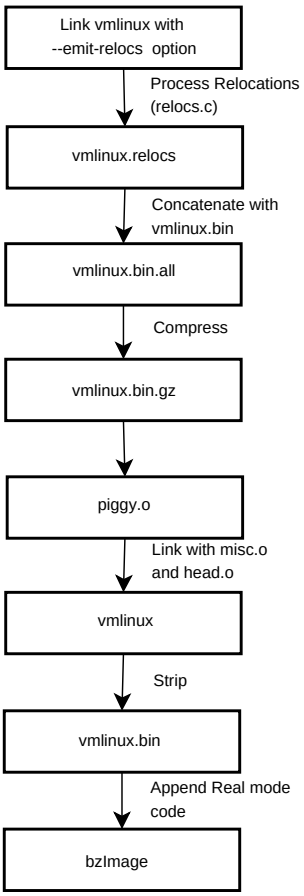


Figure 1: i386 bzImage build process

2.2.2 In-place Decompression

The code for decompressing the kernel has been changed so that decompression can be done in-place. Now the kernel is not first decompressed to any other memory location then merged and put at final destination and there are no hard-coded assumptions about the intermediate location [7]. This allows the kernel to be decompressed within the bounds of its uncompressed size and it will not overwrite any other data. Figure 2 depicts the new bzImage decompression logic.

At the same time, the decompressor is compiled as position independent code (-fPIC) so that it is not bound to a physical location, and it can run from anywhere. This code has been carefully written to make sure that it runs even if no relocation processing is done.

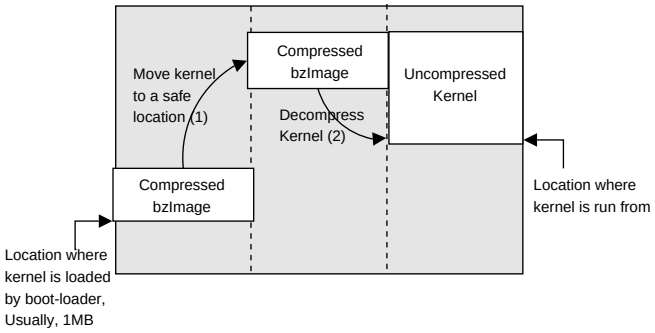


Figure 2: In-place bzImage decompression

2.2.3 Perform Relocations

After decompression, all the relocations are performed. Uncompressed code calculates the difference between the address for which the kernel was compiled and the address at which it is loaded. This difference is added to the locations as specified by relocation offsets and control is transferred to the 32-bit kernel entry point.

2.2.4 Kernel Config Options

Several new config options have been introduced for relocatable kernel implementation. `CONFIG_RELOCATABLE` controls whether the resulting bzImage is relocatable or not. If this option is not set, no relocation information is generated in the vmlinux.

Generally, bzImage decompresses itself to the address it has been compiled for (`CONFIG_PHYSICAL_START`) and runs from there. But if `CONFIG_RELOCATABLE` is set, then it runs from the address it has been loaded at by the boot-loader and it ignores the compile time address.

`CONFIG_PHYSICAL_ALIGN` option allows a user to specify the alignment restriction on the physical address the kernel is running from.

2.3 Design Details (x86_64)

In x86_64, kernel text and data are not part of the linearly mapped region and are mapped in a separate 40MB virtual address range. Hence, one can easily remap the kernel text and data region depending on where the kernel is loaded in the physical address space.

The kernel decompression logic has been changed to do an in-place decompression. The changes are similar to those as discussed for i386 architecture.

2.3.1 Kernel text And data Mapping Modification

Normally, the kernel text/data virtual and physical addresses differ by an offset of `__START_KERNEL_map` (0xffffffff80000000UL). At run time, this offset will change if the kernel is not loaded at the same address it has been compiled for. Kernel initial boot code determines where it is running from and it updates its page tables accordingly. This shift in address is calculated at run time and is stored in a variable `phys_base`. Figure 3 depicts the various mappings.

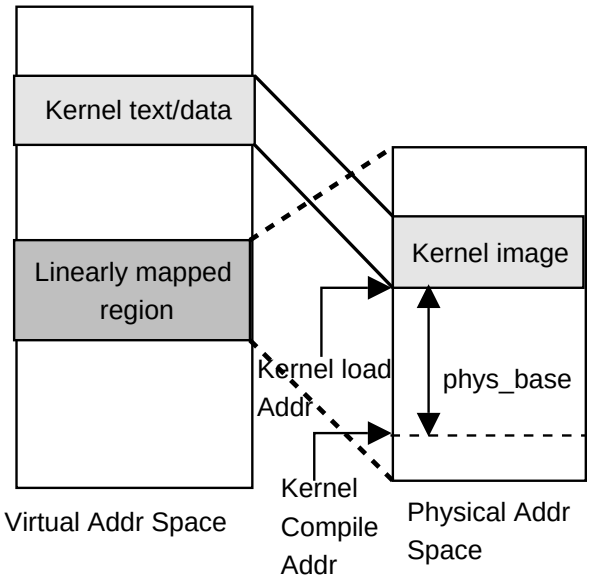


Figure 3: x86_64 kernel text/data mapping update

2.3.2 __pa_symbol() and __pa() Changes

Given the fact kernel text/data mapping changes at run time, some `__pa()`-related macro definitions need to be modified.

As mentioned previously, the kernel determines the difference between the address it has been compiled for and the address it has been loaded at and stores that shift in the variable `phys_base`.

Currently, `__pa_symbol()` is used to determine the physical address associated with a kernel text/data virtual address. Now this mapping is not fixed and can vary at run time. Hence, `__pa_symbol()` has been updated to take into the account the offset `phys_base` while calculating the physical address associated with a kernel text/data area.

```
#define __pa_symbol(x)      \
   ({unsigned long v;  \
     asm("" : "=r" (v) : "0" (x));\
     ((v - __START_KERNEL_map) + phys_base);})
```

`__pa()` should be used only for virtual addresses belonging to a linearly mapped region. Currently, this macro can map both the linearly mapped region and the kernel/text data region. But, now it has been updated to map only the kernel linearly mapped region, keeping in line with the rest of the architectures. As the kernel linearly mapped region mappings don't change because of kernel image location, `__pa()` does not have to handle the kernel load address shift (`phys_base`).

```
#define __pa(x) \
    ((unsigned long)(x) - PAGE_OFFSET)
```

2.3.3 Kernel Config Options

It is similar to i386, except that there is no option `CONFIG_PHYSICAL_ALIGN` and alignment is set to 2MB.

2.4 bzImage Protocol Extension

The bzImage protocol has been extended to communicate relocatable kernel information to the boot-loader. Two new fields, `kernel_alignment` and `relocatable_kernel`, have been added to the bzImage header. The first one specifies the physical address alignment requirement for the protected mode kernel, and the second one indicates whether this protected mode kernel is relocatable or not.

A boot-loader can look at the `relocatable_kernel` field and decide if the protected mode component should be loaded at the hard-coded 1MB address or it can be loaded at other addresses too. Kdump uses this feature and kexec boot-loader loads the relocatable bzImage at non-1MB address, in a reserved memory area.

3 Dump Filtering

On modern machines with a large amount of memory, capturing the contents of the entire RAM could create a huge dump file, which might be in the range of terabytes. It is difficult to manage such a huge dump file, both while storing it locally and while sending it to a remote host for analysis. *makedumpfile* is a userspace utility to create a smaller dump file by dump filtering, or by compressing the dump data, or both [5].

3.1 Filtering Options

Often, many pages like userspace pages, free pages, and cache pages might not be of interest to the engineer analyzing the crash dump. Hence, one can choose to filter out those pages. The following are the types of pages one can filter out:

- Pages filled with zero
  `makedumpfile` distinguishes this page type by reading each page. These pages are not part of the dump file but the analysis tool is returned zeros while accessing the filtered zero page.

- Cache pages
  `makedumpfile` distinguishes this page type by reading the members `flags` and `mapping` in `struct page`. If both the `PG_lru` bit and `PG_swapcache` bit of `flags` are on and `PAGE_MAPPING_ANON` bit of `mapping` is off, the page is considered to be a cache page.

- User process data pages
  `makedumpfile` distinguishes this page type by reading the member `mapping` in `struct page`. If `PAGE_MAPPING_ANON` bit of `mapping` is on, the page is considered to be a user process data page.

- Free pages
  `makedumpfile` distinguishes this page type by reading the member `free_area` in `struct zone`. If the page is linked into the member `free_area`, the page is considered to be a free page.

3.2 Filtering Implementation Details

`makedumpfile` examines the various memory management related data structures in the core file to distinguish between page types. It uses the crashed kernel vmlinux, compiled with debug information, to retrieve a variety of information like data structure size, member offset, symbol addresses, and so on.

The memory management information depends on the Linux version, the architecture of the processor, and the memory model (`FLATMEM`, `DISCONTIGMEM`, `SPARSEMEM`). For example, the symbol name of `struct pglist_data` is `node_data` on linux-2.6.18 x86_64 `DISCONTIGMEM`, but it is `pgdat_list` on linux-2.6.18 ia64 `DISCONTIGMEM`. `makedumpfile` supports these varieties.

To begin with, `makedumpfile` infers the memory model used by the crashed kernel, by searching for the symbol `mem_map`, `mem_section`, `node_data`, or `pgdat_list` in the binary file of the production kernel. If symbol `mem_map` is present, the crashed kernel used `FLATMEM` memory model or if `mem_section` is present, the crashed kernel used `SPARSEMEM` memory model or if `node_data` or `pgdat_list` is present, the crashed kernel used `DISCONTIGMEM` memory model.

Later it examines the `struct page` entry of each page frame and retrieves the members `flags` and `mapping`. The size of `struct page` and the member field offsets are extracted from the `.debug_info` section of the debug-compiled vmlinux of the production kernel. Various symbol virtual addresses are retrieved from the symbol table of production kernel binary.

The organization of `struct page` entry arrays, depends on the memory model used by the kernel. For the `FLATMEM` model on linux-2.6.18 i386, `makedumpfile` determines the virtual address of the symbol `mem_map` from vmlinux. This address is translated into file offset with the help of `/proc/vmcore` ELF headers and finally it reads the `mem_map` array at the calculated file offset from core file. Other page types in various memory models are distinguished in similar fashion.

3.3 Dump File Compression

The dump file can be compressed using standard compression tools like gzip to generate smaller footprint. The only drawback is that one will have to uncompress the whole file before starting the analysis. Alternatively, one can compress individual pages and decompress a particular page only when analysis tool accesses it. This reduces the disk space requirement while analyzing a crash dump.

`makedumpfile` allows for the creation of compressed dump files where compression is done on a per page basis. `diskdump` has used this feature in the past and `makedumpfile` has borrowed the idea [3].

3.4 Dump File Format

By default, `makedumpfile` creates a dump file in the kdump-compressed format. It is based on `diskdump` file format with minor modifications. The `crash` utility can analyze kdump-compressed format.

`makedumpfile` can also create a dump file in ELF format which can be opened by both `crash` and `gdb`. The ELF format does not support compressed dump files.

3.5 Sample Dump Filtering Results

The dump file size depends on the production kernel's memory usage. Tables 1 and 2 show the dump file size reduction in two possible cases. In the first table, most of the production kernel's memory is free, as dump was captured immediately after a system boot and filtering out free pages is effective. In the second table, most of the memory is used as cache, as a huge file was being decompressed while dump was captured, and filtering out cache pages is effective.

linux-2.6.18, x86_64 Memory:5GB	
Filtering option	Size Reduction
Pages filled with zero	76%
Cache pages	16%
User process data pages	1%
Free pages	78%
All the above types	97%

Table 1: Dump filtering on system containing many free pages

linux-2.6.18, x86_64 Memory:5GB	
Filtering option	Size Reduction
Pages filled with zero	3%
Cache pages	91%
User process data pages	1%
Free pages	1%
All the above types	94%

Table 2: Dump filtering on system containing many cache pages

4 Kdump initramfs

In the early days of kdump, crash dump capturing was automated with the help of init scripts in userspace. This approach was simple and easy, but it assumed that the root filesystem was not corrupted during system crash and could still be mounted safely in the second kernel. Another consideration is that one should not have to run various other init scripts before he/she starts saving the dump. Other scripts unnecessarily consume precious kernel memory and possibly can lead to reduced reliability.

These limitations led to the idea of capturing the crash dump from early userspace (initial ramdisk context). Saving the dump before even a single init script runs, probably adds to the reliability of the operation and precious memory is not consumed by un-required init scripts. Also, one could specify a dump device other than the root partition, which is guaranteed to be safe.

A prototype implementation of initrd based dumping was available in Fedora® 6. This was a basic scheme implemented along the lines of nash shell based standard boot initrd and had various drawbacks like bigger ramdisk size, limited dump destination devices supported, and limited error handling capability because of constrained scripting environment.

The above limitations triggered the redesign of the initrd based dumping mechanism. The following sections provide the details of the new design and also highlight the short-comings of the existing implementation, wherever appropriate.

4.1 Busybox-based Initial RAM Disk

Initial implementation of initrd based dumping was roughly based on the initramfs files generated by the `mkinitrd` utility. The newer design, uses Busybox [2] utilities to generate the kdump initramfs. The advantages of this scheme become evident in the following discussion.

4.1.1 Managing the initramfs Size

One of the primary design goals was to keep the initramfs as small as possible for two reasons. First, one wants to reserve as little memory as possible for crash dump kernel to boot and second, out of the reserved memory, one wants to keep the free memory pool as large as possible, to be used by kernel and drivers.

Initially it was considered to implement all of the required functionality for kdump in a statically linked binary, written in C. This binary would have been smaller than Busybox, as it would avoid inclusion of unused Busybox bits. But maintainability of the above approach was a big concern, keeping in mind the vast array of functionality it had to support. The feature list included the ability to copy files to nfs mounts, to local disk drives, to local raw partitions, and to remote servers via ssh.

Upon a deeper look, the problem space resembled more and more that of an embedded system which made the immediate solution to many of the constraints self evident: Busybox [2].

Following are some of the utilities which are typically packed into the initial ramdisk and contribute to the size bloat of initramfs.

- `nash`: A non-interactive shell-like environment
- The `cp` utility
- The `scp` and `ssh` utilities: If a scp remote target is selected
- The `ifconfig` utility
- The `dmsetup` and `lvm` utilities: For software raided and lvm file systems

Some of these utilities are already built statically. However, even if one required the utilities to be dynamically linked, various libraries have to be pulled in to satisfy dependencies and the initramfs image size skyrockets. In the case of the earlier initramfs for kdump, depending on the configuration, the inclusion of `ssh`, `scp`, `ifconfig`, and `cp` required the inclusion of the following libraries:

libacl.so.1	libz.so.1
libselinux.so.1	libnsl.so.1
libc.so.6	libcrypt.so.1
libattr.so.1	libgssapi_krb5.so.2
libdl.so.2	libkrb5.so.3
libsepol.so.1	libk5crypto.so.3
linux-gate.so.1	libcom_err.so.2
libresolv.so.2	libkrb5support.so
libcrypto.so.6	ld-linux.so.2
libutil.so.1	

Given these required utilities and libraries, the initramfs was initially between 7MB and 11MB uncompressed, which seriously cut into the available heap presented to the kernel and the userspace applications which needed it during the dump recovery process.

Busybox immediately provided a remedy to many of the size issues. By using Busybox, the cp and ifconfig utilities were no longer needed, and with them went away the need for most of the additional libraries. With Busybox, our initramfs size was reduced to a range of 2MB to 10MB.

4.1.2 Enhanced Flexibility

A Busybox based initramfs implementation vastly increased kdump system flexibility. Initially, the nash interpreter allowed us a very small degree of freedom in terms of how we could capture crash dumps. Given that nash is a non-interactive script interpreter with an extremely limited conditional handling infrastructure, we were forced in our initial implementation to determine, at initramfs creation time, exactly what our crash procedure would be. In the event of any malfunction, there was only one error handling path to choose, which was failing to capture the dump and rebooting the system.

Now, with Busybox, we are able to replace nash with any of the supported Busybox shells (msh was chosen, since it was the most bash-like shell that Fedora's Busybox build currently enables). This switch gave us several improvements right away, such as an increase in our

ability to make decisions in the init script. Instead of a script that was in effect a strictly linear progression of commands, we now had the ability to create if-then-else conditionals and loops, as well as the ability to create and reference variables in the script. We could now write an actual shell script in the initramfs, which allowed us to, among many other things, to recover from errors in a less drastic fashion. We now have the choice to do something other than simply reboot the system and lose the core file. Currently, it tries to mount the root filesystem and continue the boot process, or drop to an interactive shell within the initramfs so that a system administrator can attempt to recover the dump manually.

In fact, the switch to Busybox gave us a good deal of additional features that we were able to capitalize on. Of specific note was the additional networking ability in Busybox. The built-in ifup/ifdown network interface framework allowed us to bring network interfaces up and down easily, while the built-in vconfig utility allowed us to support remote core capture over vlan interfaces. Furthermore since we now had a truly scriptable interface, we were able to use the sysfs interface to enslave interfaces to one another, emulating the ifenslave utility, which in turn allows us to dump cores over bonded interfaces. Through the use of sysfs, we are also able to dynamically query the devices that are found at boot time and create device files for them on the fly, rather than having to anticipate them at initramfs creation time. Add to that the ability to use Busybox's findfs utility to identify local partitions by disklabel or uuid, and we are able to dynamically determine our dump location at boot time without needing to undergo a kdump initramfs rebuild every time local disk geometry changes.

4.2 Future Goals

In the recent past, our focus in terms of kdump userspace implementation has been on moving to Busybox in an effort to incorporate and advance upon the functionality offered by previous dump capture utilities, while minimizing the size impact of the initramfs and promoting maintainability. Now that we have achieved these goals, at least in part, our next set of goals include the following:

- **Cleanup initramfs generation** – The generation of the initramfs has been an evolutionary process. Current initramfs generation script is a heavily modified version of its predecessor to support the use of Busybox. This script needs to be reimplemented to be more maintainable.
- **Config file formalization** – The configuration file syntax for kdump is currently very ad hoc, and does not easily support expansion of configuration directives in any controlled manner. The configuration file syntax should be formalized.
- **Multiple dump targets** – Currently, the initramfs allows the configuration of one dump target, and a configurable failure action in the event the dump capture fails. Ideally, the configuration file should support the listing of several dump targets as alternatives in case of failures.
- **Further memory reduction** – While we have managed to reduce memory usage in the initramfs by a large amount, some configurations still require the use of large memory footprint binaries (most notably scp and ssh). Eventually, we hope to switch to using a smaller statically linked ssh client for use in remote core capture instead, to reduce the top end of our memory usage.

5 Linux Kernel Dump Test Module

Before adopting any dumping mechanism, it is important to ascertain that the solution performs reliably in most crash scenarios. To achieve this, one needed a tool which can be used to trigger crash dumps from various kernel code paths without patching and rebuilding the kernel. LKDTM (Linux Kernel Dump Test Module) is a dynamically loadable kernel module, that can be used for forcing a system crash in various scenarios and helps in evaluating the reliability of a crash dumping solution. It has been merged with the mainline kernel and is available in kernel version 2.6.19.

LKDTM is based on LKDTT (Linux Kernel Dump Test Tool) [10], but has an entirely different design. LKDTT inserts the crash points statically and one must patch and rebuild the kernel before it can be tested. On the other hand, LKDTM makes use of `jprobes` infrastructure and allows crash points to be inserted dynamically.

5.1 LKDTM Design

LKDTM artificially induces system crashes at predefined locations and triggers dump for correctness test-

ing. The goal is to widen the coverage of the tests, to take into account the different conditions in which the system might crash, for example, the state of the hardware devices, system load, and context of execution.

LKDTM achieves the crash point insertion by using jumper probes (`jprobes`), the dynamic kernel instrumentation infrastructure of the Linux kernel. The module places a *jprobe* at the entry point of a critical function. When the kernel control flow reaches the function, as shown in Figure 4, the probe causes the registered helper function to be called before the actual function is executed. At the time of insertion, each crash point is associated with two attributes: the action to be triggered and the number of times the crash point is to be hit before triggering the action (similar to LKDTT). In the helper function, if it is determined that the count associated with the crash point has been hit, the specified action is performed. The supported action types, referred to as Crash Types, are *kernel panic, oops, exception, and stack overflow*.

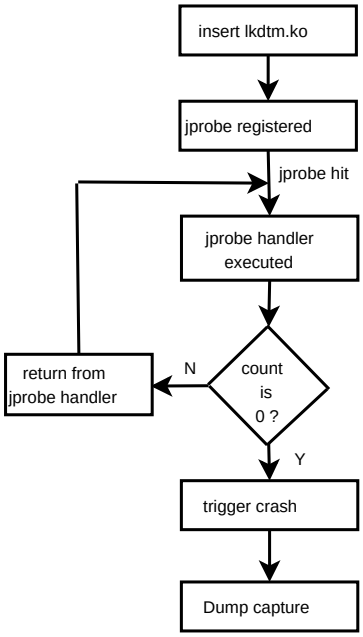


Figure 4: LKDTM functioning

`jprobes` was chosen over `kprobes` in order to ensure that the action is triggered in the same context as that of the critical function. In the case of `kprobes`, the helper function is executed in the *int 3* trap context. Whereas, when the *jprobe* is hit, the underlying `kprobes` infrastructure points the saved instruction pointer to the *jprobe's* handler routine and returns from the int3 trap (refer Documentation/kprobes.txt for the working of `kprobes/jprobes`). The helper routine is then executed in the same context as that of the critical function, thus preserving the kernel's execution mode.

5.2 Types of Crash Points

The basic crash points supported by LKDTM are same as supported by LKDTT. These are as follows:

IRQ handling with IRQs disabled (`INT_HARDWARE_ENTRY`) The *jprobe* is placed at the head of the function `__do_IRQ`, which processes interrupts with IRQs disabled.

IRQ handling with IRQs enabled (`INT_HW_IRQ_EN`) The *jprobe* is placed at the head of the function `handle_IRQ_event`, which processes interrupts with IRQs enabled.

Tasklet with IRQs disabled (`TASKLET`) This crash point recreates crashes that occur when the tasklets are being executed with interrupts disabled. The *jprobe* is placed at the function `tasklet_action`.

Block I/O (`FS_DEVRW`) This crash point crashes the system when the filesystem accesses the low-level block devices. It corresponds to the function `ll_rw_block`.

Swap-out (`MEM_SWAPOUT`) This crash point causes the system to crash while in the memory swapping is being performed.

Timer processing (`TIMERADD`) The *jprobe* is placed at function `hrtimer_start`.

SCSI command (`SCSI_DISPATCH_CMD`) This crash point is placed in the SCSI dispatch command code.

IDE command (`IDE_CORE_CP`) This crash point brings down the system while handling I/O on IDE block devices.

New crash points can be added, if required, by making changes to drivers/misc/lkdtm.c file.

5.3 Usage

The LKDTM module can be built by enabling the `CONFIG_LKDTM` config option under the `Kernel hacking` menu. It can then be inserted into the running kernel by providing the required command line arguments, as shown:

```
#modprobe lkdtm cpoint_name=<> cpoint_
type=<> [cpoint_count={>0}] [recur_
count={>0}]
```

5.4 Advantages/disadvantages of LKDTM

LKDTT has kernel space and user space components. In order to make use of LKDTT, one has to apply the kernel patch and rebuild the kernel. Also, it makes use of the Generalised Kernel Hooks Interface (GHKI), which is not part of the mainline kernel. On the other hand, using LKDTM is extremely simple and is merged into mainline kernels. The crash point can be injected into a running kernel by simply inserting the kernel module.

The only shortcoming of LKDTM is that the crash point cannot be placed in the middle of the function without changing the context of execution, unlike LKDTT.

5.5 Kdump Testing Automation

So far kdump testing was done manually but it was difficult and very time consuming process. Now it has been automated with the help of LKDTM infrastructure and some scripts.

LTP (Linux Test Project) seems to be the right place for such testing automation framework. A patch has been posted to LTP mailing list [4]. This should greatly help distributions in quickly identify regressions with every new release.

These scripts set up a cron job which starts on a reboot and inserts either LKDTM or an elementary testing module called `crasher`. Upon a crash, a crash dump is automatically captured and saved to a pre-configured location. This is repeated for various crash points as supported by LKDTM. Later, these scripts also open the captured dump and do some basic sanity verification.

The tests can be started by simply executing the following from within the tarball directory:

```
# ./setup
# ./master run
```

The detailed instructions on the usage have been documented in the README file, which is part of the tarball.

6 Device Driver Hardening

Device driver initialization in a capture kernel continues to be a pain point. Various kinds of problems have been reported. A very common problem is the pending messages/interrupts on the device from previous the kernel's context. This interrupt is delivered to the driver in the capture kernel's context and it often crashes because of state mismatch. A solution is based on the fact that the device should have a way to allow the driver to reset it. Reset should bring it to a known state from where the driver can continue to initialize the device.

PCI bus reset can probably be of help here, but it is uncertain how the PCI bus can be reset from software. There does not seem to be a generic way, but PowerPC® firmware allows doing a software reset of the PCI buses and the Extended Error Handling (EEH) infrastructure makes use of it. We are looking into using EEH functionality to reset the devices while the capture kernel boots.

Even if there is a way to reset the device, device drivers might not want to reset it all the time as resetting is generally time consuming. To resolve this issue, a new command line parameter `reset_devices` has been introduced. When this parameter is passed on the command line, it is an indication to the driver that it should first try to reset the underlying device and then go ahead with the rest of the initialization.

Some drivers like megaraid, mptfusion, ibmvscsi and ibmveth reported issues and have been fixed. MPT tries to reset the device if it is not in an appropriate state and megaraid sends a FLUSH/ABORT message to the device to flush all the commands sent from previous kernel's context. More problems have been reported with aacraid and cciss drivers which are yet to be fixed.

7 Early Boot Crash Dumping

Currently, kdump does not work if the kernel crashes before the boot process is completed. For kdump to work, the kernel should be booted up so that the new kernel is pre-loaded. During the development phase, many developers run into issues when the kernel is not booting at all, and making crash dumps work in those scenarios will be useful.

One idea is to load the kernel from early userspace (initrd/initramfs), but that does not solve the problem entirely because the kernel can crash earlier than that.

Another possibility is to use the `kboot` boot-loader to pre-load a dump capture kernel in the memory somewhere and then launch the production kernel. This production kernel will jump to the already loaded capture kernel in case of a boot time crash. Figure 5 illustrates the above design.

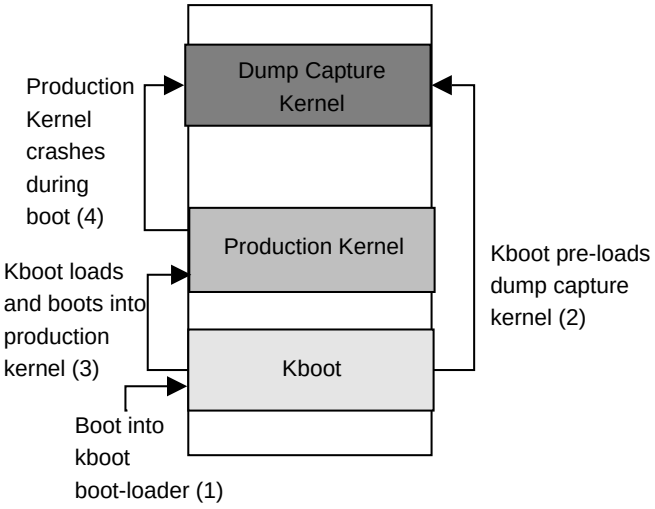


Figure 5: Early boot crash dumping

8 Conclusions

Kdump has come a long way since the initial implementation was merged into the 2.6.13 kernels. Features like the relocatable kernel, dump filtering, and initrd-based dumping have made it an even more reliable and easy to use solution. Distributions are in the process of merging these features in upcoming releases for mass deployment.

The only problem area is the device driver initialization issues in the capture kernel. Currently, these issues are being fixed on a per-driver basis when they are reported. We need more help from device driver maintainers to fix the reported issues. We are exploring the idea of performing device reset using EEH infrastructure on Power and that should further improve the reliability of kdump operation.

9 Legal Statement

References

[1] Absolute symbol relocation entries. `http://lists.gnu.org/archive/html/bug-binutils/2006-07/msg00080.html`.

[2] Busybox. `http://www.busybox.net/`.

[3] diskdump project. `http://sourceforge.net/projects/lkdump`.

[4] Ltp kdump test suite patch tarball. http://marc.theaimsgroup.com/?l=ltp-list&m=117163521109921&w=2.

[5] makedumpfile project. https://sourceforge.net/projects/makedumpfile.

[6] Jan Kratochvil's prototype implementation. http://marc.info/?l=osdl-fastboot&m=111829071126481&w=2.

[7] Werner Almsberger. Booting Linux: The History and the Future. In *Proceedings of the Linux Symposium, Ottawa*, 2000.

[8] Eric W. Biederman. Initial prototype implementation. http://marc.info/?l=linux-kernel&m=115443019026302&w=2.

[9] Eric W. Biederman. Shared library based implementation. http://marc.info/?l=osdl-fastboot&m=112022378309030&w=2.

[10] Fernando Luis Vazquez Cao. Evaluating Linux Kernel Crash Dumping Mechanisms. In *Proceedings of the Linux Symposium, Ottawa*, 2005.

[11] Vivek Goyal. x86-64 relocatable bzimage patches. http://marc.info/?l=linux-kernel&m=117325343223898&w=2.

[12] Vivek Goyal. Kdump, A Kexec Based Kernel Crash Dumping Mechanism. In *Proceedings of the Linux Symposium, Ottawa*, 2005.

Using KVM to run Xen guests without Xen

Ryan A. Harper
IBM
ryanh@us.ibm.com

Michael D. Day
IBM
ncmike@us.ibm.com

Anthony N. Liguori
IBM
aliguori@us.ibm.com

Abstract

The inclusion of the Kernel Virtual Machine (KVM) driver in Linux 2.6.20 has dramatically improved Linux's ability to act as a hypervisor. Previously, Linux was only capable of running UML guests and containers. KVM adds support for running unmodified x86 operating systems using hardware-based virtualization. The current KVM user community is limited by the availability of said hardware.

The Xen hypervisor, which can also run unmodified operating systems with hardware virtualization, introduced a paravirtual ABI for running modified operating systems such as Linux, Netware, FreeBSD, and OpenSolaris. This ABI not only provides a performance advantage over running unmodified guests, it does not require any hardware support, increasing the number of users that can utilize the technology. The modifications to the Linux kernel to support the Xen paravirtual ABI are included in some Linux distributions, but have not been merged into mainline Linux kernels.

This paper will describe the modifications to KVM required to support the Xen paravirtual ABI and a new module, **kvm-xen**, implementing the requisite hypervisor-half of the ABI. Through these modifications, Linux can add to its list of supported guests all of the paravirtual operating systems currently supported by Xen. We will also evaluate the performance of a XenoLinux guest running under the Linux hypervisor to compare Linux's ability to act as a hypervisor versus a more traditional hypervisor such as Xen.

1 Background

The x86 platform today is evolving to better support virtualization. Extensions present in both AMD and Intel's latest generation of processors include support for simplifying CPU virtualization. Both AMD and Intel plan on providing future extensions to improve MMU and hardware virtualization.

Linux has also recently gained a set of x86 virtualization enhancements. For the 2.6.20 kernel release, the `paravirt_ops` interface and KVM driver were added. `paravirt_ops` provides a common infrastructure for hooking portions of the Linux kernel that would traditionally prevent virtualization. The KVM driver adds the ability for Linux to run unmodified guests, provided that the underlying processor supports either AMD or Intel's CPU virtualization extensions.

Currently, there are three consumers of the `paravirt_ops` interface: Lguest [Lguest], a "toy" hypervisor designed to provide an example implementation; VMI [VMI], which provides support for guests running under VMware; and XenoLinux [Xen], which provides support for guests running under the Xen hypervisor.

1.1 Linux as a Hypervisor

As of 2.6.20, Linux will have the ability to virtualize itself using either hardware assistance on newer processors or paravirtualization on existing and older processors.

It is now interesting to compare Linux to existing hypervisors such as VMware [VMware] and Xen [Xen]. Some important questions include:

- What level of security and isolation does Linux provide among virtual guests?
- What are the practical advantages and disadvantages of using Linux as a hypervisor?
- What are the performance implications versus a more traditional hypervisor design such as that of Xen?

To begin answering these questions and others, we use Linux's virtualization capabilities to run a XenoLinux kernel as a guest within Linux.

The Xen hypervisor is now in several Linux distributions and forms the basis of two commercial hypervisors. Therefore, the paravirtual kernel interface that Xen supports is a good proof point of completeness and performance. If Linux can support guests well using a proven interface such as Xen's, then Linux can be a practical hypervisor.

2 Virtualization and the x86 Platform

Starting in late 2005, the historical shortcomings of the x86 platform with regard to virtualization were remedied by Intel and AMD. It will take another several years before most x86 platforms in the field have hardware virtualization support. There are two classes of problems when virtualizating older x86 platforms:

1. Functional issues, including the existence of privileged instructions that are available to non-privileged code. [Robin]
2. Performance issues that arise when multiple kernels are running on the platform.

Most of the work done by VMware and Xen, as well as the code in Linux, `paravirt_ops` and `Lguest`, is focused on overcoming these x86 shortcomings.

Performance issues caused by virtualization, including TLB flushing, cache thrashing, and the need for additional layers of memory management, are being addressed in current and future versions of Intel and AMD processors.

2.1 Ring Compression and Trap-and-Emulate

Traditionally, a monitor and a guest mode have been required in the CPU to virtualize a platform. [Popek] This allows a hypervisor to know whenever one of its guests is executing an instruction that needs hypervisor intervention, since executing a privileged instruction in guest mode will cause a trap into monitor mode.

A second requirement for virtualization is that all instructions that potentially modify the guest isolation mechanisms on the platform must trap when executed in guest mode.

Both of these requirements are violated by Intel and AMD processors released prior to 2005. Nevertheless VMware, and later Xen, both successfully virtualized these older platforms by devising new methods and working with x86 privilege levels in an unanticipated way.

The x86 processor architecture has four privilege levels for segment descriptors and two for page descriptors. *Privileged* instructions are able to modify control registers or otherwise alter the protection and isolation characteristics of segments and pages on the platform. Privileged instructions must run at the highest protection level (ring 0). All x86 processors prior to the introduction of hardware virtualization fail to generate a trap on a number of instructions when issued outside of ring 0. These "non-virtualizable" instructions prevent the traditional trap-and-emulate virtualization method from functioning.

Ring compression is the technique of loading a guest kernel in a less-privileged protection zone, usually ring 3 (the least privileged) for user space and ring 1 for the guest kernel (or ring 3 for the `x86_64` guest kernel). In this manner, every time the guest kernel executes a trappable privileged instruction the processor enters into the hypervisor (fulfilling the first requirement above, the ability to monitor the guest), giving the hypervisor full control to disallow or emulate the trapped instruction.

Ring compression *almost* works to simulate a monitor mode on x86 hardware. It is not a complete solution because some x86 instructions that should cause a trap in certain situations do not.

2.2 Paravirtualization and Binary Translation

On x86 CPUs prior to 2005 there are some cases where non-privileged instructions can alter isolation mechanisms. These include pushing or popping flags or segment registers and returning from interrupts—instructions that are safe when only one OS is running on the host, but not safe when multiple guests are running on the host. These instructions are unsafe because they do not necessarily cause a trap when executed by non-privileged code. Traditionally, hypervisors have required this ability to use traps as a monitoring mechanism.

VMware handles the problem of non-trapping privileged instructions by scanning the binary image of the kernel, looking for specific instructions [Adams] and replacing them with instructions that either trap or call into the hypervisor. The process VMware uses to replace binary instructions is similar to that used by the Java just-in-time bytecode compiler.

Xen handles this problem by modifying the OS kernel at compile time, so that the kernel never executes any non-virtualizable instructions in a manner that can violate the isolation and security of the other guests. A Xen kernel replaces sensitive instructions with register-based function calls into the Xen hypervisor. This technique is also known as *paravirtualization*.

2.3 x86 Hardware Virtualization Support

Beginning in late 2005 when Intel started shipping Intel-VT extensions, x86 processors gained the ability to have both a host mode and a guest mode. These two modes co-exist with the four levels of segment privileges and two levels of page privileges. AMD also provides processors with similar features with their AMD-V extensions. Intel-VT and AMD-V are similar in their major features and programming structure.

With Intel-VT and AMD-V, hardware ring compression and binary translation are obviated by the new hardware instructions. The first generation of Intel and AMD virtualization support remedied many of the *functional* shortcomings of x86 hardware, but not the most important *performance* shortcomings.

Second-generation virtualization support from each company addresses key performance issues by reducing unnecessary TLB flushes and cache misses and by providing multi-level memory management support directly in hardware.

KVM provides a user-space interface for using AMD-V and Intel-VT processor instructions. Interestingly, KVM does this as a loadable kernel module which turns Linux into a virtual machine monitor.

3 Linux x86 Virtualization Techniques in More Detail

To allow Linux to run paravirtual Xen guests, we use techniques that are present in Xen, KVM, and Lguest.

3.1 Avoiding TLB Flushes

x86 processors have a translation-look-aside buffer (TLB) that caches recently accessed virtual address to physical address mappings. If a virtual address is accessed that is not present in the TLB, several hundred cycles are required to determine the physical address of the memory.

Keeping the TLB hit rate as high as possible is necessary to achieve high performance. The TLB, like the instruction cache, takes advantage of *locality of reference*, or the tendency of kernel code to refer to the same memory addresses repeatedly.

Locality of reference is much reduced when a host is running more than one guest operating system. The practical effect of this is that the TLB hit rate is reduced significantly when x86 platforms are virtualized.

This problem cannot be solved by simply increasing the size of the TLB. Certain instructions force the TLB to be flushed, including hanging the address of the page table. A hypervisor would prefer to change the page table address every time it switches from guest to host mode, because doing so simplifies the transition code.

3.2 Using a Memory Hole to Avoid TLB Flushes

Devising a safe method for the hypervisor and guest to share the same page tables is the best way to prevent an automatic TLB flush every time the execution context changes from guest to host. Xen creates a memory hole immediately above the Linux kernel and resides in that hole. To prevent the Linux kernel from having access to the memory where the hypervisor is resident, Xen uses segment limits on x86_32 hardware. The hypervisor is resident in mapped memory, but the Linux kernel cannot gain access to hypervisor memory because it is beyond the limit of the segment selectors used by the kernel. A general representation of this layout is shown in Figure 1.

This method allows both Xen and paravirtual Xen guests to share the same page directory, and hence does not force a mandatory TLB flush every time the execution context changes from guest to host. The memory hole mechanism is formalized in the `paravirt_ops reserve_top_address` function.

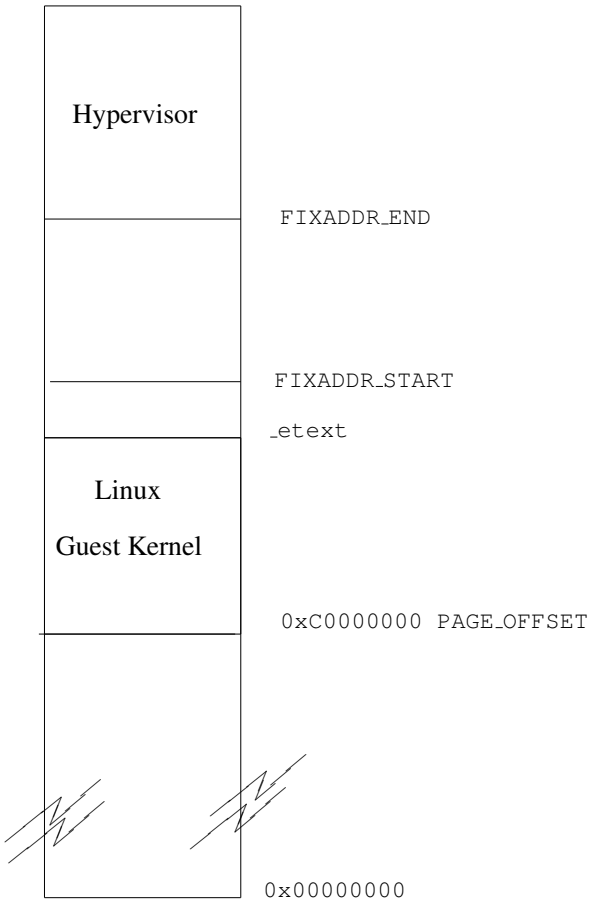


Figure 1: Using segment limits on i386 hardware allows the hypervisor to use the same page directory as each guest kernel. The guest kernel has a different segment selector with a smaller limit.

3.2.1 x86_64 hardware and TLB Flushes

The x86_64 processors initially removed segment limit checks, so avoiding TLB flushes in 64-bit works differently from 32-bit hardware. Transitions from kernel to hypervisor occur via the `syscall` instruction, which is available in 64-bit mode. `syscall` forces a change to segment selectors, dropping the privilege level from 1 to 0.

There is no memory hole *per se* in 64-bit mode, because segment limits are not universally enforced. An unfortunate consequence is that a guest's userspace must have a different page table from the guest's kernel. Therefore, the TLB is flushed on a context switch from user to kernel mode, but not from guest (kernel) to host (hypervisor) mode.

3.3 Hypercalls

A paravirtual Xen guest uses hypercalls to avoid causing traps, because a hypercall can perform much better than a trap. For example, instead of writing to a pte that is flagged as read-only, the paravirtualized kernel will pass the desired value of the pte to Xen via a hypercall. Xen will validate the new value of the pte, write to the pte, and return to the guest. (Other things happen before execution resumes at the guest, much like returning from an interrupt in Linux).

Xen hypercalls are similiar to software interrupts. They pass parameters in registers. 32-bit unprivileged guests in Xen, hypercalls are executed using an `int` instruction. In 64-bit guests, they are executed using a `syscall` instruction.

3.3.1 Handling Hypercalls

Executing a hypercall transfers control to the hypervisor running at ring 0 and using the hypervisor's stack. Xen has information about the running guest stored in several data structures. Most hypercalls are a matter of validating the requested operation and making the requested changes. Unlike instruction traps, not a lot of introspection must occur. By working carefully with page tables and processor caches, hypercalls can be much faster than traps.

3.4 The `Lguest` Monitor

`Lguest` implements a virtual machine monitor for Linux that runs on x86 processors that do not have hardware support for virtualization. A userspace utility prepares a guest image that includes an `Lguest` kernel and a small chunk of code above the kernel to perform context switching from the host kernel to the guest kernel.

`Lguest` uses the `paravirt_ops` interface to install trap handlers for the guest kernel that reflect back into `Lguest` switching code. To handle traps, `Lguest` switches to the host Linux kernel, which contains a `paravirt_ops` implementation to handle guest hypercalls.

3.5 KVM

The AMD-V and Intel-VT instructions available in newer processors provide a mechanism to force the processor to trap on certain sensitive instructions even if the processor is running in privileged mode. The processor uses a special data area, known as the VMCB or VMCS on AMD and Intel respectively, to determine which instructions to trap and how the trap should be delivered. When a VMCB or VMCS is being used, the processor is considered to be in guest mode. KVM programs the VMCB or VMCS to deliver sensitive instruction traps back into host mode. KVM uses information provided in the VMCB and VMCS to determine why the guest trapped and emulates the instruction appropriately.

Since this technique does not require any modifications to the guest operating system, it can be used to run any x86 operating that runs on a normal processor.

4 Our Work

Running a Xen guest on Linux without the Xen hypervisor present requires some basic capabilities. First, we need to be able to load a Xen guest into memory. Second, we have to initialize a Xen compatible start-of-day environment. Lastly, Linux needs to be able to switch between running the guest as well as to support handling the guest's page tables. For the purposes of this paper we are not addressing virtual IO, nor SMP; however, we do implement a simple console device. We discuss these features in Section 6. We expand on our choice of using QEMU as our mechanism to load and initialize the guest for running XenoLinux in Section 4.2. Using the Lguest switching mechanism and tracking the guest state is explained in Section 4.3. Section 4.4 describes using Linux's KVM infrastructure for Virtual Machine creation and shadowing a VM's page tables. The virtual console mechanism is explained in Section 4.5. Finally, we describe the limitations of our implementation in Section 4.6.

4.1 QEMU and Machine Types

QEMU is an open source machine emulator which uses translation or virtulization to run operating systems or programs for various machine architectures. In addition to emulating CPU architectures, QEMU also emulates platform devices. QEMU combines a CPU and a set of devices together in a QEMU machine type. One example is the "pc" machine which emulates the 32-bit x86 architecture and emulates a floppy controller, RTC, PIT, IOAPIC, PCI bus, VGA, serial, parallel, network, USB, and IDE disk devices.

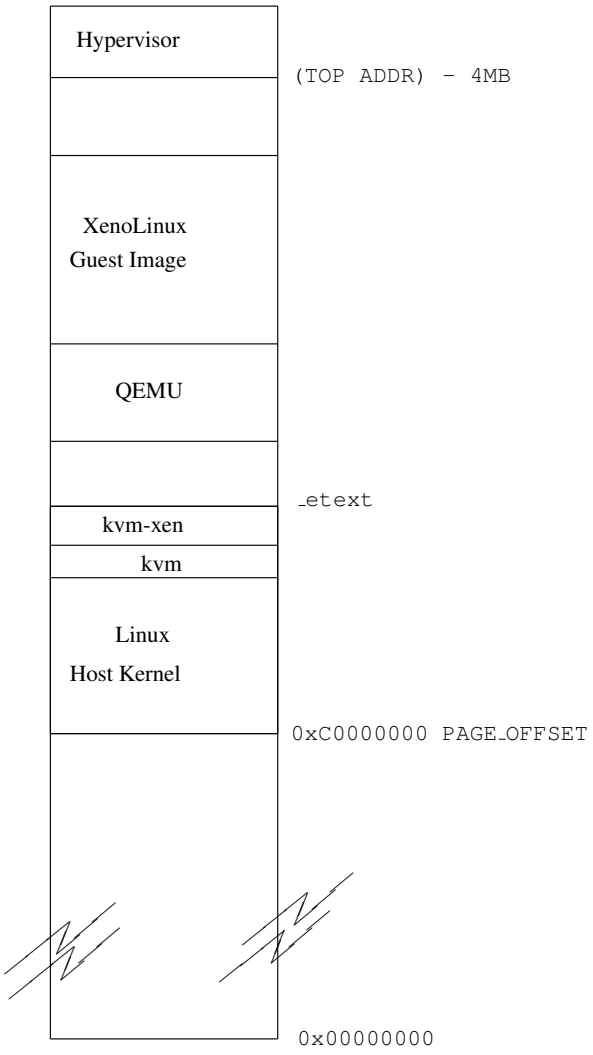


Figure 2: Linux host memory layout with KVM, **kvm-xen**, and a XenoLinux guest

A paravirtualized Xen guest can be treated as a new QEMU machine type. Specifically, it is a 32-bit CPU which only executes code in ring 1 and contains no devices. In addition to being able to define which devices are to be emulated on a QEMU machine type, we can also control the initial machine state. This control is quite useful as Xen's start-of-day assumptions are not the same as a traditional 32-bit x86 platform. In the end, the QEMU Xen machine type can be characterized as an eccentric x86 platform that does not run code in ring 0,

nor has it any hardware besides the CPU.

Paravirtualized kernels, such as XenoLinux, are built specifically to be virtualized, allowing QEMU to use an accelerator, such as KVM. KVM currently relies on the presence of hardware virtualization to provide protection while virtualizing guest operating systems. Paravirtual kernels do not need hardware support to be virtualized, but they do require assistance in transitioning control between the host and the guest. Lguest provides a simple mechanism for the transitions we explain below.

4.2 Lguest Monitor and Hypercalls

Lguest is a simple x86 paravirtual hypervisor designed to exercise the `paravirt_ops` interface in Linux 2.6.20. No hypercalls are implemented within Lguest's hypervisor, but instead it will transfer control to the host to handle the requested work. This delegation of work ensures a very small and simple hypervisor. Paravirtual Xen guests rely on hypercalls to request that some work be done on its behalf.

For our initial implementation, we chose not to implement the Xen hypercalls in our hypervisor directly, but instead reflect the hypercalls to the QEMU Xen machine. Handling a Xen hypercall is fairly straightforward. When the guest issues a hypercall, we examine the register state to determine which hypercall was issued. If the hypercall is handled in-kernel (as some should be for performance reasons) then we simply call the handler and return to the guest when done. If the hypercall handler is not implemented in-kernel, we transition control to the QEMU Xen machine. This transition is done by setting up specific PIO operations in the guest state and exiting `kvm_run`. The QEMU Xen machine will handle the hypercalls and resume running the guest.

4.3 KVM Back-end

The KVM infrastructure does more than provide an interface for using new hardware virtualization features. The KVM interface gives Linux a generic mechanism for constructing a VM using kernel resources. We needed to create a new KVM back-end which provides access to the infrastructure without requiring hardware support.

Our back-end for KVM is **kvm-xen**. As with other KVM back-ends such as kvm-intel and kvm-amd, **kvm-xen** is required to provide an implementation of the `struct kvm_arch_ops`. Many of the arch ops are designed to abstract the hardware virtualization implementation. This allows **kvm-xen** to provide its own method for getting, setting, and storing guest register state, as well as hooking on guest state changes to enforce protection with software in situations where kvm-intel or kvm-amd would rely on hardware virtualization.

We chose to re-use Lguest's structures for tracking guest state. This choice was obvious after deciding to re-use Lguest's hypervisor for handling the transition from host to guest. Lguest's hypervisor represents the bare minimum needed in a virtual machine monitor. For our initial work we saw no compelling reason to write our own version of the switching code.

During our hardware setup we map the hypervisor into the host virtual address space. We suffer the same restrictions on the availability of a specific virtual address range due to Lguest's assumption that on most machines the top 4 megabytes are unused. During VCPU creation, we allocate a `struct lguest`, reserve space for guest state, and allocate a trap page.

After the VCPU is created, KVM initializes its shadow MMU code. Using the `vcpu_setup` hook which fires after the MMU is initialized, we set up the initial guest state. This setup involves building the guest GDT, IDT, TSS, and the segment registers.

When `set_cr3` is invoked the KVM MMU resets the shadow page table and calls into back-end specific code. In **kvm-xen**, we use this hook to ensure that the hypervisor pages are always mapped into the shadow page tables and to ensure that the guest cannot modify those pages.

We use a modified version of Lguest's `run_guest` routine when supporting the `kvm_run` call. Lguest's `run_guest` will execute the guest code until it traps back to the host. Upon returning to the host, it decode the trap information and decides how to proceed. **kvm-xen** follows the same model, but replaces Lguest-specific responses such as replacing Lguest hypercalls with Xen hypercalls.

4.4 Virtual Console

The virtual console as expected by a XenoLinux guest is a simple ring queue. Xen guests expect a reference to a page of memory shared between the guest and the

host and a Xen event channel number. The QEMU Xen machine type allocates a page of the guest's memory, selects an event channel, and initializes the XenoLinux start-of-day with these values.

During the guest booting process it will start writing console output to the shared page and will issue a hypercall to notify the host of a pending event. After control is transferred from the guest, the QEMU Xen machine examines which event was triggered. For console events, the QEMU Xen machine reads data from the ring queue on the shared page, increments the read index pointer, and notifies the guest that it received the message via the event channel bitmap, which generates an interrupt in the guest upon returning from the host. The data read from and written to the shared console page is connected to a QEMU character device. QEMU exposes the character devices to users in many different ways including telnet, Unix socket, PTY, and TCP socket. In a similar manner, any writes to QEMU character devices will put data into the shared console page, increment the write index pointer, and notify the guest of the event.

4.5 Current Restrictions

The current implementation for running Xen guests on top of Linux via KVM supports 32-bit UP guests. We have not attempted to implement any of the infrastructure required for virtual IO beyond simple console support. Additionally, while using Lguest's hypervisor simplified our initial work, we do inherit the requirement that the top 4 megabytes of virtual address space be available on the host. We discuss virtual IO and SMP issues as future work in Section 6.

5 Xen vs. Linux as a hypervisor

One of the driving forces behind our work was to compare a more traditional microkernel-based hypervisor with a hypervisor based on a monolithic kernel. **kvm-xen** allows a XenoLinux guest to run with Linux as the hypervisor allowing us to compare this environment to a XenoLinux guest running under the Xen hypervisor. For our evaluation, we chose three areas to focus on: security, manageability, and performance.

5.1 Security

A popular metric to use when evaluating how secure a system can be is the size of the Trusted Computing Base (or TCB). On a system secured with static or dynamic attestation, it is no longer possible to load arbitrary privileged code [Farris]. This means the system's security is entirely based on the privileged code that is being trusted.

Many claim that a microkernel-based hypervisor, such as Xen, significantly reduces the TCB since the hypervisor itself is typically much smaller than a traditional operating system [Qiang]. The Xen hypervisor would appear to confirm this claim when we consider its size relative to an Operating System such as Linux.

Project	*SLOCs*
KVM	8,950
Xen	165,689
Linux	5,500,933

Figure 3: Naive TCB comparison of KVM, Xen, and Linux

From Figure 3, we can see that Xen is thirty-three times smaller than Linux. However, this naive comparison is based on the assumption that a guest running under Xen does not run at the same privilege level as Xen itself. When examining the TCB of a Xen system, we also have to consider any domain that is privileged. In every Xen deployment, there is at least one privileged domain, typically Domain-0, that has access to physical hardware. Any domain that has access to physical hardware has, in reality, full access to the entire system. The vast majority of x86 platforms do not possess an IOMMU which means that every device capable of performing DMA can access any region of physical memory. While privileged domains do run in a lesser ring, since they can program hardware to write arbitrary data to arbitrary memory, they can very easily escalate their privileges.

When considering the TCB of Xen, we must also consider the privileged code running in any privileged domain.

Project	*SLOCs*
KVM	5,500,933
Xen	5,666,622

Figure 4: TCB size of KVM and Xen factoring in the size of Linux

We clearly see from Figure 4 that since the TCB of both

kvm-xen and Xen include Linux, the TCB comparison really reduces down to the size of the **kvm-xen** module versus the size of the Xen hypervisor. In this case, we can see that the size of the Xen TCB is over an order magnitude larger than **kvm-xen**.

5.2 Driver Domains

While most x86 platforms do not contain IOMMUs, both Intel and AMD are working on integrating IOMMU functionality into their next generation platforms [VT-d]. If we assume that eventually, IOMMUs will be common for the x86, one could argue that Xen has an advantage since it could more easily support driver domains.

5.3 Guest security model

The Xen hypervisor provides no security model for restricting guest operations. Instead, any management functionality is simply restricted to root from the privileged domain. This simplistic model requires all management software to run as root and provides no way to restrict a user's access to a particular set of guests.

kvm-xen, on the other hand, inherits the Linux user security model. Every **kvm-xen** guest appears as a process which means that it also is tied to a UID and GID. A major advantage of **kvm-xen** is that the supporting software that is needed for each guest can be run with non-root privileges. Consider the recent vulnerability in Xen related to the integrated VNC server [CVE]. This vulnerability actually occurred in QEMU which is shared between KVM and Xen. It was only a security issue in Xen, though, as the VNC server runs as root. In KVM, the integrated VNC server runs as a lesser user, giving the VM access only to files on the host that are accessible by its user.

Perhaps the most important characteristic of the process security model for virtualization is that it is well understood. A Linux administrator will not have to learn all that much to understand how to secure a system using **kvm-xen**. This reduced learning curve will inherently result in a more secure deployment.

5.4 Tangibility

Virtualization has the potential to greatly complicate machine management, since it adds an additional layer of abstraction. While some researchers are proposing new models to simplify virtualization [Sotomayor], we believe that applying existing management models to virtualization is an effective way to address the problem.

The general deployment model of Xen is rather complicated. It first requires deploying the Xen hypervisor which must boot in place of the operating system. A special kernel is then required to boot the privileged guest–Domain-0. There is no guarantee that device drivers will work under this new kernel, although the vast majority do. A number of key features of modern Linux kernels are also not available such as frequency scaling and software suspend. Additionally, regardless of whether any guests are running, Xen will reserve a certain amount of memory for the hypervisor—typically around 64MB.

kvm-xen, on the other hand, is considerably less intrusive. No changes are required to a Linux install when **kvm-xen** is not in use. A special host kernel is not needed and no memory is reserved. To deploy **kvm-xen**, one simply needs to load the appropriate kernel module.

Besides the obvious ease-of-use advantage of **kvm-xen**, the fact that it requires no resources when not being used means that it can be present on any Linux installation. There is no trade-off, other than some disk space, to having **kvm-xen** installed. This lower barrier to entry means that third parties can more easily depend on a Linux-based virtualization solution such as **kvm-xen** than a microkernel-based solution like Xen.

Another benefit of **kvm-xen** is that is leverages the full infrastructure of QEMU. QEMU provides an integrated VNC server, a rich set of virtual disk formats, userspace virtual network, and many other features. It is considerably easier to implement these features in QEMU since it is a single process. Every added piece of infrastructure in Xen requires creating a complex communication protocol and dealing with all sorts of race conditions.

Under **kvm-xen**, every XenoLinux guest is a process. This means that the standard tools for working with processes can be applied to **kvm-xen** guests. This greatly simplifies management as eliminates the need to create and learn a whole new set of tools.

5.5 Performance

At this early stage in our work, we cannot definitively answer the question of whether a XenoLinux guest un-

Project	*Cycles*
xen-pv	154
kvm-xen	1151
kvm-svm	2696

Figure 5: Hypercall latency

der **kvm-xen** will perform as well or better than running under the Xen hypervisor. We can, however, use our work to attempt to determine whether the virtualization model that **kvm-xen** uses is fundamentally less performant than the model employed by Xen. Further, we can begin to determine how significant that theoretical performance difference would be.

The only fundamental difference between **kvm-xen** and the Xen hypervisor is the cost of a hypercall. With the appropriate amount of optimization, just about every other characteristic can be made equivalent between the two architectures. Hypercall performance is rather important in a virtualized environment as most of the privileged operations are replaced with hypercalls.

As we previously discussed, since the Xen Hypervisor is microkernel-based, the virtual address space it requires can be reduced to a small enough amount that it can fit within the same address space as the guest. This means that a hypercall consists of only a privilege transition. Due to the nature of x86 virtualization, this privilege transition is much more expensive than a typical syscall, but is still relatively cheap.

Since **kvm-xen** uses Linux as its hypervisor, it has to use a small monitor to trampoline hypercalls from a guest to the host. This is due to the fact that Linux cannot be made to fit into the small virtual address space hole that the guest provides. Trampolining the hypercalls involves changing the virtual address space and, subsequently, requires a TLB flush. While there has been a great deal of work done on the performance impact of this sort of transition [Wiggins], for the purposes of this paper we will attempt to consider the worst-case scenario.

In the above table, we see that **kvm-xen** hypercalls are considerably worse than Xen hypercalls. We also note though that **kvm-xen** hypercalls are actually better than hypercalls when using SVM. Current SVM-capable processors require an address space change on every world switch so these results are not surprising.

Based on these results, we can assume that **kvm-xen** should be able to at least perform as well as an SVM guest can today. We also know from many sources [XenSource] that SVM guests can perform rather well on many workloads, suggesting that **kvm-xen** should also perform well on these workloads.

6 Future Work

To take kvm-xen beyond our initial work, we must address how to handle Xen's virtual IO subsystem, SMP capable guests, and hypervisor performance.

6.1 Xen Virtual IO

A fully functional Xen virtual IO subsystem is comprised of several components. The XenoLinux kernel includes a virtual disk and network driver built on top of a virtual bus (Xenbus), an inter-domain page-sharing mechanism (grant tables), and a data persistence layer (Xenstore). For kvm-xen to utilize the existing support in a XenoLinux kernel, we need to implement support for each of these elements.

The Xenbus element is mostly contained within the XenoLinux guest, not requiring significant work to be utilized by kvm-xen. Xenbus is driven by interaction with Xenstore. As changes occur within the data tracked by Xenstore, Xenbus triggers events within the XenoLinux kernel. At a minimum, kvm-xen needs to implement the device enumeration protocol in Xenstore so that XenoLinux guests have access to virtual disk and network.

Xen's grant-tables infrastructure is used for controlling how one guest shares pages with other domains. As with Xen's Domain 0, the QEMU Xen machine is also capable of accessing all of the guest's memory, removing the need to reproduce grant-table-like functionality.

Xenstore is a general-purpose, hierarchical data persistence layer. Its implementation relies on Linux notifier chains to trigger events with a XenoLinux kernel. kvm-xen would rely on implementing a subset of Xenstore functionality in the QEMU Xen machine.

6.2 SMP Guests

Providing support for XenoLinux SMP guests will be very difficult. As of this writing, KVM itself does not

support SMP guests. In addition to requiring KVM to become SMP capable, XenoLinux kernels rely on the Xen hypervisor to keep all physical CPU Time Stamp Counter (TSC) registers in relative synchronization. Linux currently does not utilize TSCs in such a fashion using other more reliable time sources such as ACPI PM timers.

6.3 Hypervisor Performance

Xen guests that utilize shadow page tables benefit significantly from the fact that the shadow paging mechanism is within the hypervisor itself. kvm-xen uses KVM's MMU, which resides in the host kernel, and XenoLinux guests running on kvm-xen would benefit greatly from moving the MMU into the hypervisor. Additionally, significant performance improvements would be expected from moving MMU and context-switch-related hypercalls out of the QEMU Xen machine and into the hypervisor.

7 Conclusion

With **kvm-xen** we have demonstrated that is possible to run a XenoLinux guest with Linux as its hypervisor. While the overall performance picture of running XenoLinux guests is not complete, our initial results indicate that **kvm-xen** can achieve adequate performance without using a dedicated microkernel-based hypervisor like Xen. There are still some significant challenges for **kvm-xen**—namely SMP guest support—though as KVM and the `paravirt_ops` interface in Linux evolve, implementing SMP support will become easier.

8 References

[Lguest] Russell, Rusty. *lguest (formerly lhype).* Ozlabs. 2007. 10 Apr. 2007 `http://lguest.ozlabs.org`.

[VMI] Amsden, Z. *Paravirtualization API Version 2.5.* VMware. 2006.

[Xen] Barham P., et al. *Xen and the art of virtualization.* In Proc. SOSP 2003. Bolton Landing, New York, U.S.A. Oct 19–22, 2003.

[VMware] `http://www.vmware.com`

[Robin] Robin, J. and Irvine, C. *Analysis of Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*. Proceedings of the 9th USENIX Security Symposium, Denver, CO, August 2000.

[Popek] Popek, G. and Goldberg, R. *Formal Requirements for Virtualizable Third Generation Architectures.* Communications of the ACM, July 1974.

[Adams] Adams, K. and Agesen, O. *A Comparison of Software and Hardware Techniques for x86 Virtualization.* ASPLOS, 2006

[Farris] Farris, J. *Remote Attestation.* University of Illinois at Urbana-Champaign. 6 Dec. 2005.

[Qiang] Qiang, H. *Security architecture of trusted virtual machine monitor for trusted computing.* Wuhan University Journal of Natural Science. Wuhan University Journals Press. 15 May 2006.

[VT-d] Abramson, D.; Jackson, J.; Muthrasanallur, S.; Neiger, G.; Regnier, G.; Sankaran, R.; Schoinas, I.; Uhlig, R.; Vembu, B.; Wiegert, J. *Intel® Virtualization Technology for Directed I/O.* Intel Technology Journal. `http://www.intel.com/technology/itj/2006/v10i3/` (August 2006).

[CVE] *Xen QEMU Vnc Server Arbitrary Information Disclosure Vunerability.* CVE-2007-0998. 14 Mar. 2007 `http://www.securityfocus.com/bid/22967`

[Sotomayor] Sotomayor, B. (2007). *A Resource Management Model for VM-Based Virtual Workspaces*. Unpublished masters thesis, University of Chicago, Chicago, Illinois, United States.

[Wiggins] Wiggins, A., et al. *Implementation of Fast Address-Space Switching and TLB Sharing on the StrongARM Processor.* University of New South Wales.

[XenSource] *A Performance Comparision of Commercial Hypervisors.* XenSource. 2007. `http://blogs.xensource.com/rogerk/wp-content/uploads/2007/03/hypervisor_performance_comparison_1_0_5_with_esx-data.pdf`

Djprobe—Kernel probing with the smallest overhead

Masami Hiramatsu
Hitachi, Ltd., Systems Development Lab.
masami.hiramatsu.pt@hitachi.com

Satoshi Oshima
Hitachi, Ltd., Systems Development Lab.
satoshi.oshima.fk@hitachi.com

Abstract

Direct Jump Probe (djprobe) is an enhancement to kprobe, the existing facility that uses breakpoints to create probes anywhere in the kernel. Djprobe inserts jump instructions instead of breakpoints, thus reducing the overhead of probing. Even though the kprobe "booster" speeds up probing, there still is too much overhead due to probing to allow for the tracing of tens of thousands of events per second without affecting performance.

This presentation will show how the djprobe is designed to insert a jump, discuss the safety of insertion, and describe how the cross self-modification (and so on) is checked. This presentation also provides details on how to use djprobe to speed up probing and shows the performance improvement of djprobe compared to kprobe and kprobe-booster.

1 Introduction

1.1 Background

For the use of non-stop servers, we have to support a probing feature, because it is sometimes the only method to analyze problematic situations.

Since version 2.6.9, Linux has kprobes as a very unique probing mechanism [7]. In kprobe ((a) in Figure 1), an original instruction at probe point is copied to an *out-of-line buffer* and a break-point instruction is put at the probe point. The break-point instruction triggers a break-point exception and invokes `pre_handler()` of the kprobe from the break-point exception handler. After that, it executes the out-of-line buffer in single-step mode. Then, it triggers a single-step exception and invokes `post_handler()`. Finally, it returns to the instruction following the probe point.

This probing mechanism is useful. For example, system administrators may like to know why their system's performance is not very good under heavy load. Moreover, system-support venders may like to know why their system crashed by salvaging traced data from the dumped memory image. In both cases, if the overhead due to probing is high, it will affect the result of the measurment and reduce the performance of applications. Therefore, it is preferable that the overhead of probing becomes as small as possible.

From our previous measurement [10] two years ago, the processing time of kprobe was about 1.0 usec whereas Linux Kernel State Tracer (LKST) [2] was less than 0.1 usec. From our previous study of LKST [9], about 3% of overhead for tracing events was recorded. Therefore, we decided our target for probing overhead should be less than 0.1 usec.

1.2 Previous Works

Figure 1 illustrates how the probing behaviors are different among kprobe, kprobe-booster and djprobe when a process hits a probe point.

As above stated, our goal of the processing time is less than 0.1 usec. Thus we searched for improvements to reduce the probing overhead so as it was as small as possible. We focused on the probing process of kprobe, which causes exceptions twice when each probe hit. We predicted that most of the overhead came from the exceptions, and we could reduce it by using jumps instead of the exceptions.

We developed the kprobe-booster as shown in Figure 1(b). In this improvement, we attempted to replace the single-step exception with a jump instruction, because it was easier than replacing a break-point. Thus, the first-half of processing of kprobe-booster is same as the kprobe, but it does not use a single-step exception in the latter-half. This improvement has already been merged into upstream kernel since 2.6.17.

Last year, Ananth's paper [7] unveiled efforts for improving kprobes, so the probing overheads of kprobe

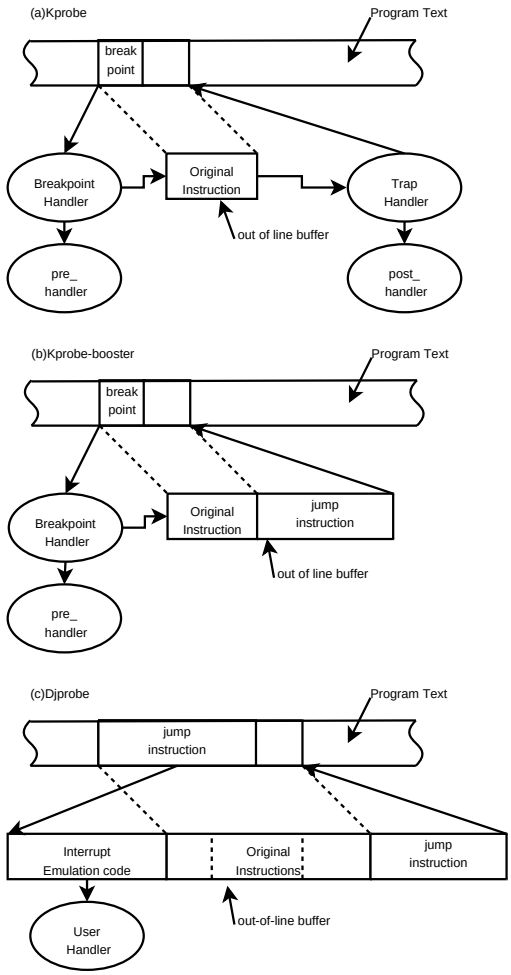


Figure 1: Kprobe, kprobe-booster and djprobe

and kprobe-booster became about 0.5 usec and about 0.3 usec respectively. Thus the kprobe-booster succeeded to reduce the probing overhead by almost half. However, its performance is not enough for our target.

Thus, we started developing djprobe: *Direct Jump Probe*.

1.3 Concept and Issues

The basic idea of djprobe is simply to use a jump instruction instead of a break-point exception. In djprobe ((c) in Figure 1), a process which hits a probe point jumps to the out-of-line buffer, calls probing handler, executes the "original instructions" on the out-of-line buffer directly, and jumps back to the instruction following the place where the original instructions existed. We will see the result of this improvement in Section 4.

There are several difficulties to implement this concept. A jump instruction must occupy 5 bytes on i386, replacement with a jump instruction changes the instruction boundary, original instructions are executed on another place, and these are done on the running kernel. So...

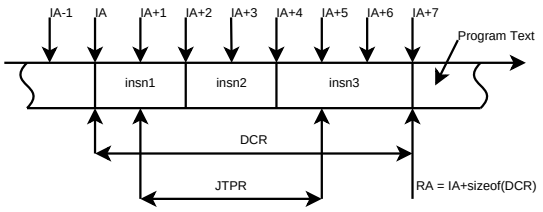
- Replacement of original instructions with a jump instruction must not block other threads.
- Replacement of original instructions which are targeted by jumps must not cause unexpectable crashes.
- Some instructions such as an instruction with relative addressing mode can not be executed at out-of-line buffer.
- There must be at least one instruction following the replaced original instructions to allow for the returning from the probe.
- Cross code modification in SMP environment may cause General Protection Fault by Intel Erratum.
- Djprobe (and also kprobe-booster) does not support the `post_handler`.

Obviously, some tricks are required to make this concept real. This paper discribes how djprobe solve these issues.

1.4 Terminology

Before discussing details of djprobe, we would like to introduce some useful terms. Figure 2 illustrates an example of execution code in CISC architecture. The first instruction is a 2 byte instruction, second is also 2 bytes, and third is 3 bytes.

In this paper, IA means *Insertion Address*, which specifies the address of a probe point. DCR means *Detoured Code Region*, which is a region from insertion address to the end of a detoured code. The detoured code consists of the instructions which are partially or fully covered by a jump instruction of djprobe. JTPR means *Jump Target Prohibition Region*, which is a 4 bytes (on i386) length region, starts from the next address of IA. And, RA means *Return Address*, which points the instruction next to the DCR.



ins1: 1st Instruction
ins2: 2nd Instruction
ins3: 3rd Instruction
IA: Insertion Address
RA: Return Address
JTPR: Jump Target Prohibition Region
DCR: Detoured Code Region

Figure 2: Terminology

2 Solutions of the Issues

In this section, we discuss how we solve the issues of djprobe. The issues that are mentioned above can be categorized as follows.

- Static-Analysis Issues
- Dynamic-Safety Issue
- Cross Self-modifying Issue
- Functionality-Performance Tradeoff Issue

The following section deeply discuss how to solve the issues of djprobe.

2.1 Static Analysis Issues

First, we will discuss a safety check before the probe is inserted. Djprobe is an enhancement of kprobes and it based on implementation of Kprobes. Therefore, it includes every limitation of kprobes, which means djprobe cannot probe where kprobes cannot. As figure 3 shows, the DCR may include several instructions because the size of jump instruction is more than one byte (relative jump instruction size is 5 bytes in i386 architecture). In addition, there are only a few choices of remedies at execution time because "out-of-line execution" is done directly (which means single step mode is not used). This

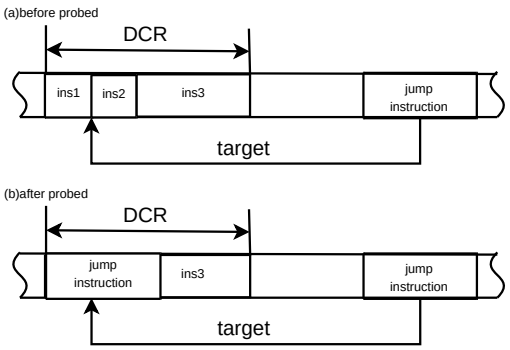


Figure 3: Corruption by jump into JTPR

means there are 4 issues that should be checked statically. See below. Static safety check must be done before registering the probe and it is enough that it be done just once. Djprobe requires that a user must not insert a probe, if the probe point doesn't pass safety checks. They must use kprobe instead of djprobe at the point.

2.1.1 Jumping in JTPR Issue

Replacement with a jump instruction involves changing instruction boundaries. Therefore, we have to ensure that no jump or call instructions in the kernel or kernel modules target JTPR. For this safety check, we assume that other functions never jump into the code other than the entry point of the function. This assumption is basically true in gcc. An exception is setjmp()/longjmp(). Therefore, djprobe cannot put a probe where setjmp() is used. Based on this assumption, we can check whether JTPR is targeted or not by looking through within the function. This code analysis must be changed if the assumption is not met. Moreover, there is no effective way to check for assembler code currently.

2.1.2 IP Relative Addressing Mode Issue

If the original instructions in DCR include the IP (Instruction Pointer, `EIP` in i386) relative addressing mode instruction, it causes the problem because the original instruction is copied to out-of-line buffer and is executed directly. The effective address of IP relative addressing mode is determined by where the instruction is placed. Therefore, such instructions will require a correction of a relative address. The problem is that almost all relative jump instructions are "near jumps" which means

destination must be within –128 to 127 bytes. However, out-of-line buffer is always farther than 128 bytes. Thus, the safety check disassembles the probe point and checks whether IP relative instruction is included. If the IP relative address is found, the djprobe can not be used.

2.1.3 Prohibit Instructions in JTPR Issue

There are some instructions that cannot be probed by djprobe. For example, a call instruction is prohibited. When a thread calls a function in JTPR, the address in the JTPR is pushed on the stack. Before the thread returns, if a probe is inserted at the point, `ret` instruction triggers a corruption because instruction boundary has been changed. The safety check also disassembles the probe point and check whether prohibited instructions are included.

2.1.4 Function Boundary Issue

Djprobe requires at least one instruction must follow DCR. If DCR is beyond the end of the function, there is no space left in the out-of-line buffer to jump back from. This safety check can easily be done because what we have to do is only to compare DCR bottom address and function bottom address.

2.2 Dynamic-Safety Issue

Next, we discuss the safety of modifying multiple instructions when the kernel is running. The dynamic-safety issue is a kind of atomicity issue. We have to take care of interrupts, other threads, and other processors, because we can not modify multiple instructions atomically. This issue becomes more serious on the preemptive kernel.

2.2.1 Simultaneous Execution Issue

Djprobe has to overwrite several instructions by a jump instruction, since i386 instruction set is CISC. Even if we write this jump atomically, there might be other threads running on the middle of the instructions which will be overwritten by the jump. Thus, "atomic write" can not help us in this situation. In contrast, we can write the break-point instruction atomically because its size is one byte. In other words, the break-point instruction modifies only one instruction. Therefore, we decided to use the "bypass method" for embedding a jump instruction.

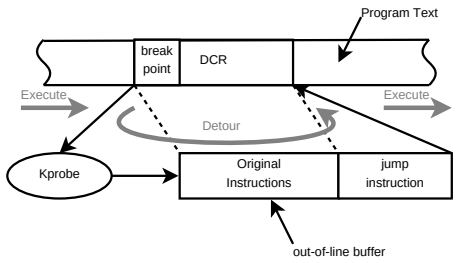


Figure 4: Bypass method

Figure 4 illustrates how this bypass method works.

This method is similar to the highway construction. The highway department makes a bypass route which detours around the construction area, because the traffic can not be stopped. Similarly, since the entire system also can not be stopped, djprobe generates an out-of-line code as a bypass from the original code, and uses a break-point of the kprobe to switch the execution address from the original address to the out-of-line code. In addition, djprobe adds a jump instruction in the end of the out-of-line code to go back to RA. In this way, other threads detour the region which is overwritten by a jump instruction while djprobe do it.

What we have to think of next is when the other threads execute from within the detoured region. In the case of non-preemptive kernel, these threads might be interrupted within the DCR. The same issue occurs when we release the out-of-line buffers. Since some threads might be running or be interrupted on the out-of-line code, we have to wait until those return from there. As you know, in the case of the non-preemptive kernel, interrupted kernel threads never call scheduler. Thus, to solve this issue, we decided to use the scheduler synchronization. Since the `synchronize_sched()` function waits until the scheduler is invoked on all processors, we can ensure all interrupts, which were occurred before calling this function, finished.

2.2.2 Simultaneous Execution Issue on Preemptive Kernel

This wait-on-`synchronize_sched` method is premised on the fact that the kernel is never preempted. In the preemptive kernel, we must use another function to wait the all threads sleep on the known places, because some threads may be preempted on the DCR. We discussed this issue deeply and decided to use the `freeze_processes()` recommended by Ingo Molnar [6]. This function tries to freeze all active processes including all preempted threads. So, preempted threads wake up and run after they call the `try_to_freeze()` or the `refrigerator()`. Therefore, if the `freeze_processe()` succeeds, all threads are sleeping on the `refrigerator()` function, which is a known place.

2.3 Cross Self-modifying Issue

The last issue is related to a processor specific erratum. The Intel® processor has an erratum about unsynchronized cross-modifying code [4]. On SMP machine, if one processor modifies the code while another processor pre-fetches unmodified version of the code, unpredictable General Protection Faults will occur. We supposed this might occur as a result of hitting a cache-line boundary. On the i386 architecture, the instructions which are bigger than 2 bytes may be across the cache-line boundary. These instructions will be pre-fetched twice from 2nd cache. Since a break-point instruction will change just a one byte, it is pre-fetched at once. Other bigger instructions, like a long jump, will be across the cache-alignment and will cause an unexpected fault. In this erratum, if the other processors issue a serialization such as `CPUID`, the cache is serialized and the cross-modifying is safely done.

Therefore, after writing the break-point, we do not write the whole of the jump code at once. Instead of that, we write only the jump address next to the break-point. And then we issue the `CPUID` on each processor by using `IPI` (Inter Processor Interrupt). At this point, the cache of each processor is serialized. After that, we overwrite the break-point by a jump op-code whose size is just one byte.

2.4 Functionality-Performance Tradeoff Issue

From Figure 1, djprobe (and kprobe-booster) does not call `post_handler()`. We thought that is a trade-off between speed and the `post_handler`. Fortunately, the SystemTap [3], which we were assuming as the main use of kprobe and djprobe, did not use `post_handler`. Thus, we decided to choose speed rather than the `post_handler` support.

3 Design and Implementation

Djprobe was originally designed as a wrapper routine of kprobes. Recently, it was re-designed as a jump optimization functionality of kprobes.[1] This section explains the latest design of djprobe on i386 architecture.

3.1 Data Structures

To integrate djprobe into kprobes, we introduce the `length` field in the `kprobe` data structure to specify the size of the DCR in bytes. We also introduce `djprobe_instance` data structure, which has three fields: `kp`, `list`, and `stub`. The `kp` field is a `kprobe` that is embedded in the `djprobe_instance` data structure. The `list` is a `list_head` for registration and unregistration. The `stub` is an `arch_djprobe_stub` data structure to hold a out-of-line buffer.

From the viewpoint of users, a `djprobe_instance` looks like a special aggregator probe, which aggregates several probes on the same probe point. This means that a user does not specify the `djprobe_instance` data structure directly. Instead, the user sets a valid value to the `length` field of a `kprobe`, and registers that. Then, that `kprobe` is treated as an aggregated probe on a `djprobe_instance`. This allows you to use djprobe transparently as a kprobe. Figure 5 illustrates these data structures.

3.2 Static Code Analysis

Djprobe requires the safety checks, that were discussed in Section 2.1, before calling `register_kprobe()`. Static code analysis tools, `djprobe_static_code_analyzer`, is available from djprobe development site [5]. This tool also provides the length of DCR. Static code analysis is done as follows.

[1] For this reason, djprobe is also known as *jump optimized kprobe*.

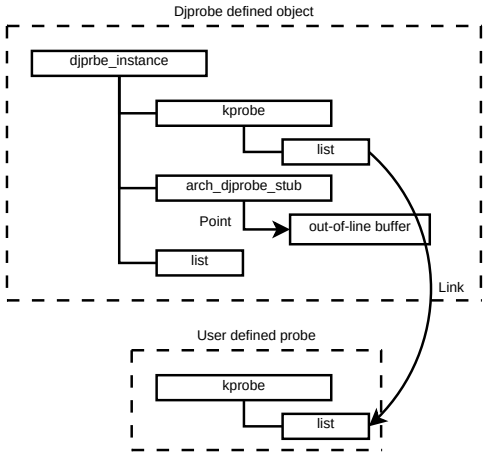


Figure 5: The instance of djprobe

```
[37af1b] subprogram
         sibling           [37af47]
         external          yes
         name              "register_kprobe"
         decl_file         1
         decl_line         874
         prototyped        yes
         type              [3726a0]
         low_pc            0xc0312c3e
         high_pc           0xc0312c46
         frame_base        2 byte block
          [   0] breg4 4
```

Figure 6: Example of debuginfo output by eu-readelf

3.2.1 Function Bottom Check

`djprobe_static_code_analyzer` requires a debuginfo file for the probe target kernel or module. It is provided by the kernel compilation option if you use vanilla kernel. Or it is provided as debuginfo package in the distro.

Figure 6 shows what debuginfo looks like.

First of all, this tool uses the debuginfo file to find the top (low_pc) and bottom (high_pc) addresses of the function where the probe point is included, and makes a list of these addresses. By using this list, it can check whether the DCR bottom exceeds function bottom. If it finds this to be true, it returns 0 as "can't probe at this point."

There are two exceptions to the function bottom check. If the DCR includes an absolute jump instruction or a function return instruction, and the last byte of these instructions equals the bottom of the function, the point can be probed by djprobe, because direct execution of those instructions sets IP to valid place in the kernel and there is no need to jump back.

3.2.2 Jump in JTPR Check

Next, `djprobe_static_code_analyzer` disassembles the probed function of the kernel (or the module) by using `objdump` tool. The problem is the current version of `objdump` cannot correctly disassemble if the `BUG()` macro is included in the function. In that case, it simply discards the output following the `BUG()` macro and retries to disassemble from right after the `BUG()`. This disassembly provides not only the boundaries information in DCR but also the assembler code in the function.

Then, it checks that all of jump or call instructions in the function do not target JTPR. It returns 0, if it find an instruction target JTPR.

If the probe instruction is 5 bytes or more, it simply returns the length of probed instruction, because there is no boundary change in JTPR.

3.2.3 Prohibited Relative Addressing Mode and Instruction Check

`djprobe_static_code_analyzer` checks that DCR does not include a relative jump instruction or prohibited instructions.

3.2.4 Length of DCR

Djprobe requires the length of DCR as an argument of `register_kprobe()` because djprobe does not have a disassembler in the current implementation. `djprobe_static_code_analyzer` acquires it and returns the length in case that the probe point passes all checks above.

3.3 Registration Procedure

This is done by calling the `register_kprobe()` function. Before that, a user must set the address of a probe point and the length[2] of DCR.

[2]If the `length` field of a kprobe is cleared, it is not treated as a djprobe but a kprobe.

3.3.1 Checking Conflict with Other Probes

First, `register_kprobe()` checks whether other probes are already inserted on the DCR of the specified probe point or not. These conflicts can be classified in following three cases.

1. Some other probes are already inserted in the same probe point. In this case, `register_kprobe()` treats the specified probe as one of the collocated probes. Currently, if the probe which previously inserted is not djprobe, the jump optimization is not executed. This behavior should be improved to do jump optimization when feasible.
2. The DCR of another djprobe covers the specified probe point. In this case, currently, this function just returns `-EEXIST`. However, ideally, the djprobe inserted previously should be unoptimized for making room for the specified probe.
3. There are some other probes in the DCR of the specified djprobe. In this case, the specified djprobe becomes a normal kprobe. This means the `length` field of the `kprobe` is cleared.

3.3.2 Creating New djprobe_instance Object

Next, `register_kprobe()` calls the `register_djprobe()` function. It allocates a `djprobe_instance` object. This function copies the values of `addr` field and `length` field from the original kprobe to the `kp` field of the `djprobe_instance`. Then, it also sets the address of the `djprobe_pre_handler()` to the `pre_handler` field of the `kp` field in the `djprobe_instance`. Then, it invokes the `arch_prepare_djprobe_instance()` function to prepare an out-of-line buffer in the `stub` field.

3.3.3 Preparing the Out-of-line Buffer

Figure 7 illustrates how an out-of-line buffer is composed.

The `arch_prepare_djprobe_instance()` allocates a piece of executable memory for the out-of-line buffer by using `__get_insn_slot()` and setup its contents. Since the original `__get_insn_slot()` function can handle only single size of memory slots, we modified it to handle various length memory slots. After allocating the buffer, it copies the template code of the buffer from the `djprobe_template_holder()` and embeds the address of the `djprobe_instance` object and the `djprobe_callback()` function into the template. It also copies the original code in the DCR of the specified probe to the next to the template code. Finally, it adds the jump code which returns to the next address of the DCR and calls `flush_icache_range()` to synchronize i-cache.

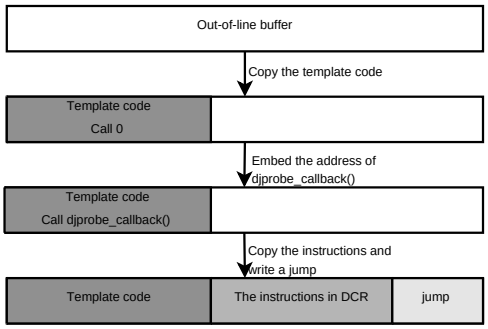


Figure 7: Preparing an out-of-line buffer

3.3.4 Register the djprobe_instance Object

After calling `arch_prepare_djprobe_instance()`, `register_djprobe()` registers the `kp` field of the `djprobe_instance` by using `__register_kprobe_core()`, and adds the `list` field to the `registering_list` global list. Finally, it adds the user-defined kprobe to the `djprobe_instance` by using the `register_aggr_kprobe()` and returns.

3.4 Committing Procedure

This is done by calling the `commit_djprobes()` function, which is called from `commit_kprobes()`.

3.4.1 Check Dynamic Safety

The `commit_djprobes()` calls the `check_safety()` function to check safety of dynamic-self modifying. In other words, it ensures that

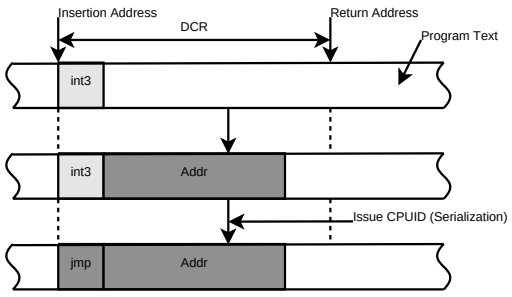


Figure 8: Optimization Procedure

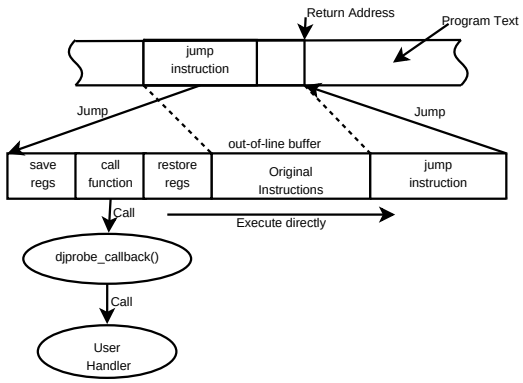


Figure 9: Probing Procedure

no thread is running on the DCR nor is it preempted. For this purpose, `check_safety()` call `synchronize_sched()` if the kernel is non-preemptive, and `freeze_processes()` and `thaw_processes()` if the kernel is preemptive. These functions may take a long time to return, so we call `check_safety()` only once.

3.4.2 Jump Optimization

Jump optimization is done by calling the `arch_preoptimize_djprobe_instance()` and the `arch_optimize_djprobe_instance()`. The `commit_djprobes()` invokes the former function to write the destination address (in other words, the address of the out-of-line buffer) into the JTPR of the djprobe, and issues `CPUID` on every online CPU. After that, it invokes the latter function to change the break-point instruction of the kprobe to the jump instruction. Figure 8 illustrates how the instructions around the insertion address are modified.

3.4.3 Cleanup Probes

After optimizing registered djprobes, the `commit_djprobe()` releases the instances of the djprobe in the `unregistering_list` list. These instances are linked by calling `unregister_kprobe()` as described Section 3.6. Since the other threads might be running on the out-of-line buffer as described in the Section 2.2, we can not release it in the `unregister_kprobe()`. However, the `commit_djprobe()` already ensured safety by using the `check_safety()`. Thus we can release the instances and the out-of-line buffers safely.

3.5 Probing Procedure

Figure 9 illustrates what happens when a process hits a probe point.

When a process hits a probe point, it jumps to the out-of-line buffer of a djprobe. And it emulates the break-point on the first-half of the buffer. This is accomplished by saving the registers on the stack and calling the `djprobe_callback()` to call the user-defined handlers related to this probe point. After that, djprobe restores the saved registers, directly executes continuing several instructions copied from the DCR, and jumps back to the RA which is the next address of the DCR.

3.6 Unregistration Procedure

This is done by calling `unregister_kprobe()`. Unlike the registration procedure, un-optimization is done in the unregistration procedure.

3.6.1 Checking Whether the Probe Is Djprobe

First, the `unregister_kprobe()` checks whether the specified kprobe is one of collocated kprobes. If it is the last kprobe of the collocated kprobes which are aggregated on a aggregator probe, it also tries to remove the aggregator. As described above, the `djprobe_instance` is a kind of the aggregator probe. Therefore, the function also checks whether the aggregator is djprobe (this is done by comparing the `pre_handler` field and the address of `djprobe_pre_handler()`). If so it calls `unoptimize_djprobe()` to remove the jump instruction written by the djprobe.

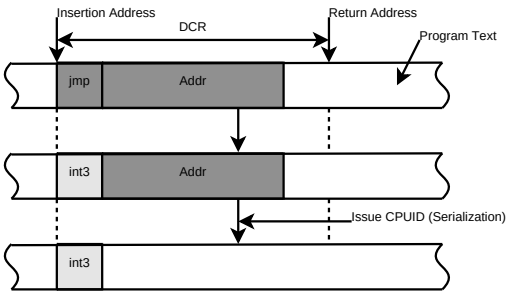


Figure 10: Un-optimization Procedure

method	orignal	booster	djprobe
kprobe	563	248	49
kretprobe	718	405	211

Table 1: Probing Time on Pentium® M in nsec

method	orignal	booster	djprobe
kprobe	739	302	61
kretprobe	989	558	312

Table 2: Probing Time on Core™ Duo in nsec

3.6.2 Probe Un-optimization

Figure 10 illustrates how a probe point is un-optimized.

The `unoptimized_djprobe()` invokes `arch_unoptimize_djprobe_instance()` to restore the original instructions to the DCR. First, it inserts a break-point to IA for protect the DCR from other threads, and issues `CPUID` on every online CPUs by using `IPI` for cache serialization. After that, it copies the bytes of original instructions to the JTPR. At this point, the djprobe becomes just a kprobe, this means it is un-optimized and uses a break-point instead of a jump.

3.6.3 Removing the Break-Point

After calling `unoptimize_djprobe()`, the `unregister_kprobe()` calls `arch_disarm_kprobe()` to remove the break-point of the kprobe, and waits on `synchronize_sched()` for cpu serialization. After that, it tries to release the aggregator if it is not a djprobe. If the aggregator is a djprobe, it just calls `unregister_djprobe()` to add the `list` field of the djprobe to the `unregistering_list` global list.

4 Performance Gains

We measured and compared the performance of djprobe and kprobes. Table 1 and Table 2 show the processing time of one probing of kprobe, kretprobe, its boosters, and djprobes. The unit of measure is nano-seconds. We measured it on Intel® Pentium® M 1600MHz with UP kernel, and on Intel® Core™ Duo 1667MHz with SMP kernel by using `linux-2.6.21-rc4-mm1`.

We can see djprobe could reduce the probing overhead to less than 0.1 usec (100 nsec) on each processor. Thus, it achived our target performance. Moreover, kretprobe can also be accelerated by djprobe, and the djprobe-based kretprobe is as fast as kprobe-booster.

5 Example of Djprobe

Here is an example of djprobe. The differences between kprobe and djprobe can be seen at two points: setting the `length` field of a `kprobe` object before registration, and calling `commit_kprobes()` after registration and unregistration.

```
/* djprobe_ex.c -- Direct Jump Probe Example */
#include <linux/version.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kprobes.h>

static long addr=0;
module_param(addr, long, 0444);
static long size=0;
module_param(size, long, 0444);

static int probe_func(struct kprobe *kp,
                      struct pt_regs *regs) {
   printk("probe point:%p\n", (void*)kp->addr);
   return 0;
}

static struct kprobe kp;

static int install_probe(void) {
   if (addr == 0) return -EINVAL;

   memset(&kp, sizeof(struct kprobe), 0);
   kp.pre_handler = probe_func;
   kp.addr = (void *)addr;

   kp.length = size;

   if (register_kprobe(&kp) != 0) return -1;

   commit_kprobes();
```

```
   return 0;
}

static void uninstall_probe(void) {
   unregister_kprobe(&kp);

   commit_kprobes();
}

module_init(install_probe);
module_exit(uninstall_probe);
MODULE_LICENSE("GPL");
```

6 Conclusion

In this paper, we proposed djprobe as a faster probing method, discussed what the issues are, and how djprobe can solve them. After that, we described the design and the implementation of djprobe to prove that our proposal can be implemented. Finally, we showed the performance improvement, and that it could reduce the probing overhead dramatically. You can download the latest patch set of djprobe from djprobe development site [5]. Any comments and contributions are welcome.

7 Future Works

We have some plans about future djprobe development.

7.1 SystemTap Enhancement

We have a plan to integrate the static analysis tool into the SystemTap for accelerating kernel probing by using djprobe.

7.2 Dynamic Code Modifier

Currently, djprobe just copies original instructions from DCR. This is the main reason why the djprobe cannot probe the place where the DCR is including execution-address-sensitive code.

If djprobe analyzes these sensitive codes and replaces its parameter to execute it on the out-of-line buffer, the djprobe can treat those codes. This idea is basically done by kerninst [1, 11] and GILK [8].

7.3 Porting to Other Architectures

Current version of djprobe supports only i386 architecture. Development for x86_64 is being considered. Several difficulties are already found, such as RIP relative instructions. In x86_64 architecuture, RIP relative addressing mode is expanded and we must assume it might be used. Related to dynamic code modifier, djprobe must modify the effective address of RIP relative addressing instructions.

To realize this, djprobe requires instruction boundary information in DCR to recognize every instruction. This should be provided by `djprobe_static_code_analyser` or djprobe must have essential version of disassembler in it.

7.4 Evaluating on the Xen Kernel

In the Xen kernel, djprobe has bigger advantage than on normal kernel, because it does not cause any interrupts. In the Xen hypervisor, break-point interruption switches a VM to the hypervisor and the hypervisor upcalls the break-point handler of the VM. This procedure is so heavy that the probing time becomes almost double.

In contrast, djprobe does not switch the VM. Thus, we are expecting the probing overhead of djprobe might be much smaller than kprobes.

8 Acknowledgments

We would like to thank Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, Maneesh Soni, Frank Ch. Eigler, Ingo Molnar, and other developers of kprobes and SystemTap for helping us develop the djprobe.

We also would like to thank Hideo Aoki, Yumiko Sugita and our colleagues for reviewing this paper.

9 Legal Statements

References

[1] kerninst. http://www.paradyn.org/html/kerninst.html.

[2] Lkst. http://lkst.sourceforge.net/.

[3] SystemTap. http://sourceware.org/systemtap/.

[4] Unsynchronized cross-modifying code operation can cause unexpected instruction execution results. Intel Pentium II Processor Specification Update.

[5] Djprobe development site, 2005. http://lkst.sourceforge.net/djprobe.html.

[6] Re: djprobes status, 2006. http://sourceware.org/ml/systemtap/2006-q3/msg00518.html.

[7] Ananth N. Mavimalayanahalli et al. Probing the Guts of Kprobes. In *Proceedings of Ottawa Linux Symposium*, 2006.

[8] David J. Pearce et al. Gilk: A dynamic instrumentation tool for the linux kernel. In *Computer Performance Evaluation/TOOLS*, 2002.

[9] Toshiaki Arai et al. Linux Kernel Status Tracer for Kernel Debugging. In *Proceedings of Software Engineering and Applications*, 2005.

[10] Masami Hiramatsu. Overhead evaluation about kprobes and djprobe, 2005. http://lkst.sourceforge.net/docs/probe-eval-report.pdf.

[11] Ariel Tamches and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *OSDI*, February 1999.

Desktop integration of Bluetooth

Marcel Holtmann
BlueZ Project
marcel@holtmann.org

Abstract

The BlueZ project has enhanced the Bluetooth implementation for Linux over the last two years. It now seamlessly integrates with D-Bus and provides a really simple and easy to use interface for the UI applications. The current API covers all needed Bluetooth core functionalities and allows running the same daemons on all Linux distributions, the Maemo or OpenMoko frameworks, and other embedded systems. The user interface is the only difference between all these systems. This allows GNOME and KDE applications to share the same list of remote Bluetooth devices and many more common settings. As a result of this, the changes to integrate Bluetooth within the UI guidelines of Maemo or OpenMoko are really small. In return, all Maemo and OpenMoko users help by fixing bugs for the Linux desktop distributions like Fedora, Ubuntu, etc., and vice versa.

1 Introduction

The desktop integration of Bluetooth technology has always been a great challenge since the Linux kernel was extended with Bluetooth support. For a long time, most of the Bluetooth applications were command-line utilities only. With the D-Bus interface for the BlueZ protocol stack, it became possible to write desktop-independent applications. This D-Bus interface has been explicitly designed for use by desktop and embedded UI applications (see Figure 1).

For the desktop integration of Bluetooth, three main applications are needed:

- Bluetooth status applet;
- Bluetooth properties dialog;
- Bluetooth device wizard.

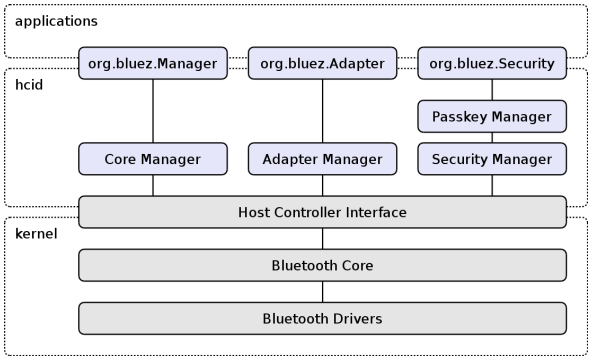


Figure 1: D-Bus API overview

2 Bluetooth applet

The Bluetooth applet is the main entry point when it comes to device configuration and handling of security-related interactions with the user, like the input of a PIN code.

One of the simple tasks of the applet is to display a Bluetooth icon that reflects the current status of the Bluetooth system such as whether a device discovery is in progress, or a connection has been established, and so on. It is up to the desktop UI design guidelines to decide if the icon itself should change or if notification messages should be displayed to inform the user of status changes.

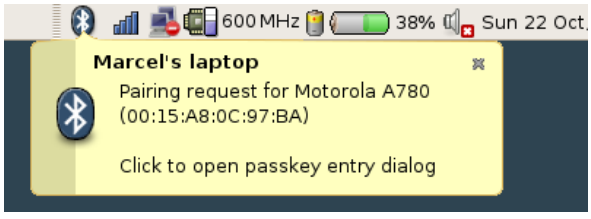


Figure 2: Bluetooth applet notification

Besides the visible icon, the applet has to implement the default passkey and authorization agent interfaces.

These two interfaces are used to communicate with the Bluetooth core daemon. The task of the applet is to display dialog boxes for requesting PIN codes or authorization question to the end user. The input will be handed back to the daemon which then actually interacts with the Bluetooth hardware.

Additionally, the applet might provide shortcuts for frequently used Bluetooth tasks. An example would be the launch of the Bluetooth configuration dialog or device setup wizard.

Figure 2 shows the notification of a pairing request for the GNOME Bluetooth applet.

3 Bluetooth properties

While the applet shows the current status of the Bluetooth system and handles the security related tasks, the properties dialog can be used to configure the local Bluetooth adapter (see Figure 3).

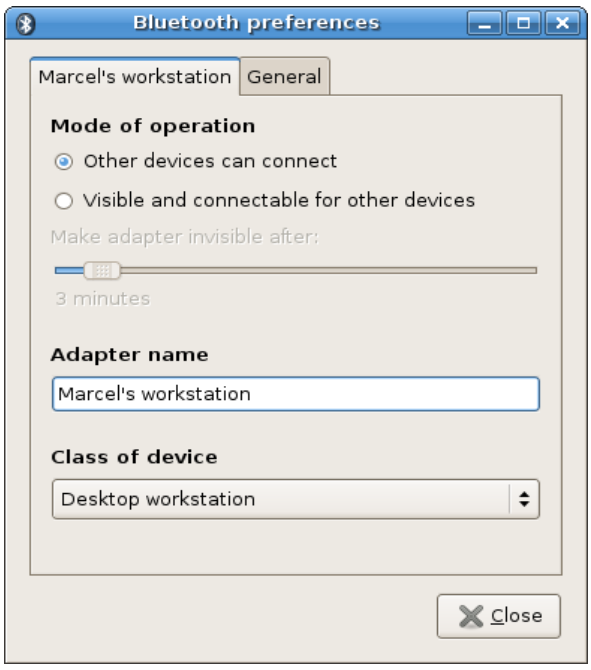


Figure 3: Bluetooth adapter configuration

The D-Bus interface restricts the possible configurable options to the local adapter name, class of device, and mode of operation. No additional options have been found useful. The Bluetooth core daemon can adapt other options automatically when needed.

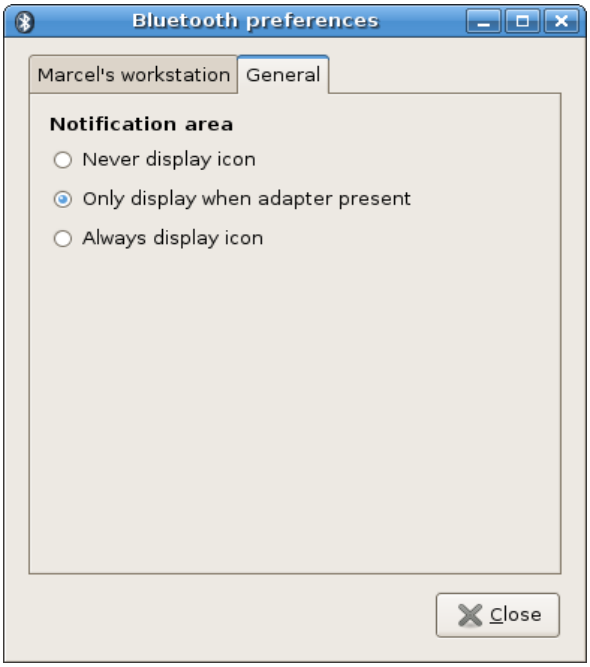


Figure 4: Bluetooth adapter configuration

In addition to the Bluetooth adapter configuration, the Bluetooth properties application can also control the behavior of the applet application (see Figure 4)—for example, the visibility of the Bluetooth status icon. It is possible to hide the icon until an interaction with the user is required.

These additional configuration options are desktop- and user-specific. The GNOME desktop might implement them differently than KDE.

4 Bluetooth wizard

With the status applet and the properties dialog, the desktop task for user interaction, notification, and the general adapter configuration are covered. The missing task is the setup of new devices. The Bluetooth wizard provides an automated process to scan for devices in range and setup any discovered devices to make them usable for the user (see Figure 5).

The wizard uses the basic Bluetooth core adapter interface to search for remote devices in range. Then, it presents the user a list of possible devices filtered by the class of device. After device selection, the wizard tries to automatically setup the services. For these tasks it uses special Bluetooth service daemons.

Currently the Bluetooth subsystem provides the following service daemons that can be used by the wizard or any other Bluetooth application:

- Network service
  - PAN support (NAP, GN and PANU)
  - LAN access (work in progress)
  - ISDN dialup (work in progress)
- Input service
  - HID support (report mode with recent kernel versions)
  - Emulated input devices (headset and proprietary protocols)
  - Wii-mote and PlayStation3 Remote
- Audio service
  - Headset and Handsfree support
  - High quality audio support (work in progress)
- Serial service
  - Emulated serial ports

Figure 5: Bluetooth device selection

5 Big picture

The BlueZ architecture has grown rapidly and the whole system became really complex. Figure 6 shows a simplified diagram of the current interactions between the Bluetooth subsystem of the Linux kernel, the Bluetooth core daemons and services, and the user interface applications.

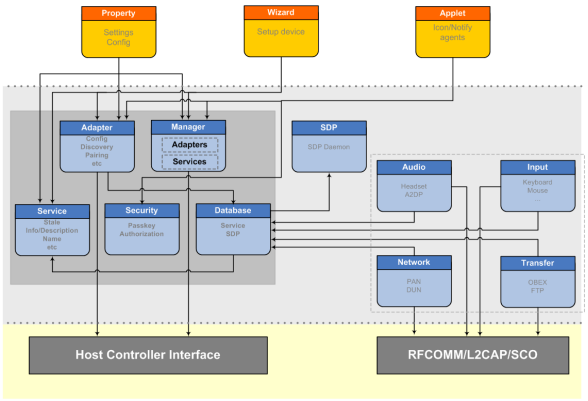


Figure 6: BlueZ architecture

All communication between daemons and a user application are done via D-Bus. Figure 7 gives an overview on how this interaction and communication via D-Bus works.

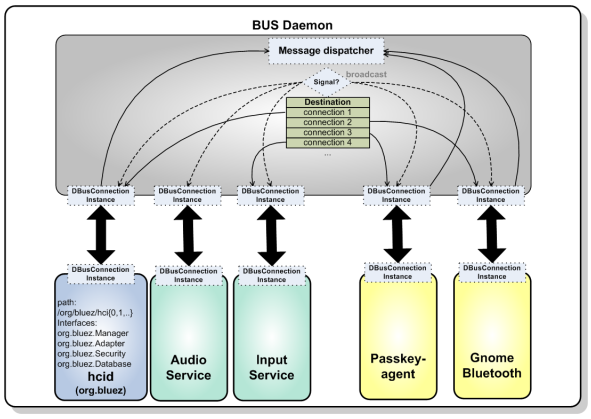


Figure 7: D-Bus communication

6 Conclusion

The *bluez-gnome* project provides an implementation for all three major Bluetooth applications needed by a modern GNOME desktop. For KDE 4, a similar set of

applications exists that uses the same D-Bus infrastructure for Bluetooth. A KDE 3 backport is currently not planned.

The desktop applications don't have to deal with any Bluetooth low-level interfaces. These are nicely abstracted through D-Bus. This allows other desktop or embedded frameworks like Maemo or OpenMoko to replace the *look and feel* quite easily (see Figure 6).

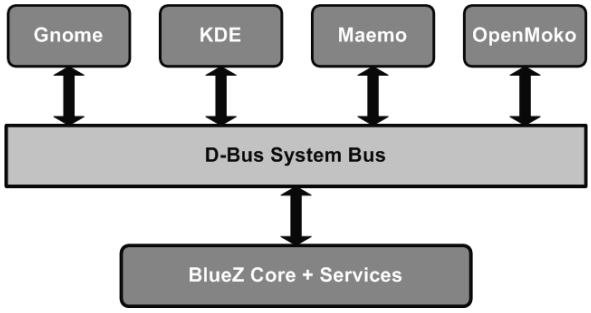


Figure 8: User interface separation

The goal of the BlueZ Project is to unify desktop and embedded Bluetooth solutions. While the user interface might be different, the actual protocol and service implementation will be the same on each system.

References

[1] Special Interest Group Bluetooth:
Bluetooth Core Specification Version 2.0 + EDR,
November 2004

[2] freedesktop.org:
D-BUS Specification Version 0.11

How virtualization makes power management different

Kevin Tian, Ke Yu, Jun Nakajima, and Winston Wang
Intel Open Source Technology Center
{kevin.tian,ke.yu,jun.nakajima,winston.l.wang}@intel.com

Abstract

Unlike when running a native OS, power management activity has different types in a virtualization world: virtual and real. A virtual activity is limited to a virtual machine and has no effect on real power. For example, virtual S3 sleep only puts the virtual machine into sleep state, while other virtual machines may still work. On the other hand, a real activity operates on the physical hardware and saves real power. Since the virtual activity is well controlled by the guest OS, the remaining problem is how to determine the real activity according to the virtual activity. There are several approaches for this problem.

1. Purely based on the virtual activity. Virtual Sx state support is a good example. Real S3 sleep will be executed if and only if all the virtual S3 are executed.
2. Purely based on the global information, regardless of the virtual activity. For example, CPU Px state can be determined by the global CPU utilization.
3. Combination of (1) and (2): in some environments, VM can directly access physical hardware with assists from hardware (e.g., Intel Virtualization Technology for Directed I/O, a.k.a. VT-d). In this case, the combination of (1) and (2) will be better.

This paper first presents the overview of power management in virtualization. Then it describes how each power management state (Sx/Cx/Px) can be handled in a virtualization environment by utilizing the above approaches. Finally, the paper reviews the current status and future work.

1 Overview of Power Management in Virtualization

This section introduces the ACPI power management state and virtualization mode, and later the overview of a power management implementation in virtualization.

1.1 Power Management state in ACPI

ACPI [1] is an open industry specification on power management and is well supported in Linux 2.6, so this paper focuses on the power management states defined in the ACPI specification.

ACPI defines several kinds of power management state:

- **Global System state (G-state)**: they are: *G0* (working), *G1* (sleeping), *G2* (soft-off) and *G3* (mechanical-off).
- **Processor Power state (C-state)**: in the *G0* state, the CPU has several sub-states, $C0 \sim Cn$. The CPU is working in *C0*, and stops working in $C1 \sim Cn$. $C1 \sim Cx$ differs in power saving and entry/exit latency. The deeper the C-state, the more power saving and the longer latency a system can get.
- **Processor Performance state (P-state)**: again, in *C0* state, there are several sub-CPU performance states (P-States). In P-states, the CPU is working, but CPU voltage and frequency vary. The P-state is a very important power-saving feature.
- **Processor Throttling state (T-state)**: T-state is also a sub state of *C0*. It saves power by only changing CPU frequency. T-state is usually used to handle thermal event.
- **Sleeping state**: In *G1* state, it is divided into several sub state: $S1 \sim S4$. They differs in power saving, context preserving and sleep/wakeup latency. *S1* is lightweight sleep, with only CPU caches lost. *S2* is not supported currently. S3 has all context lost except system memory. S4 save context to disk and then lost all context. Deeper S-state is, more power saving and the longer latency system can get.

- **Device states (D-state)**: ACPI also defines power state for devices, i.e. *D0* $\sim$ *D3*. *D0* is working state and *D3* is power-off state. *D1* and *D2* are between *D0* and *D3*. *D0* $\sim$ *D3* differs in power saving, device context preserving and entry/exit latency.

Figure 1 in Len's paper [2] clearly illustrates the state relationship.

1.2 Virtualization model

Virtualization software is emerging in the open source world. Different virtualization model may have different implementation on power management, so it is better to check the virtualization model as below.

- **Hypervisor model**: virtual machine monitor (VMM) is a new layer below operation system and owns all the hardware. VMM not only needs to provide the normal virtualization functionality, e.g. CPU virtualization, memory virtualization, but also needs to provide the I/O device driver for every device.
- **Host-based model**: VMM is built upon a host operating system. All the platform hardware including CPU, memory, and I/O device, is owned by the host OS. In this model, VMM usually exists as a kernel module and can leverage much of the host OS functionality, e.g. I/O device driver, scheduler. Current KVM is host-based model.
- **Hybrid model**: VMM is a thin layer compared to the hypervisor model, which only covers basic virtualization functionality (CPU, memory, etc.), and leave I/O device to a privileged VM. This privileged VM provides I/O service to other VM through inter-VM communication. Xen [3] is hybrid model.

Meanwhile, with some I/O virtualization technology introduced, e.g. Intel® Virtualization Technology for Directed I/O, aka VT-d, the I/O device can be directly owned by a virtual machine.

1.3 Power Management in Virtualization

Power Management in virtualization basically has the following two types:

- **Virtual power management**: this means the power management within the virtual machine. This power management only has effects to the power state of VM and does not affect other VM or hypervisor/host OS. For example, virtual S3 sleep only brings VM into virtual sleep state, while hypervisor/host OS and other VM is still working. Virtual power management usually does not save real power, but sometimes it can affect real power management.
- **Real power management**: this means the power management in the global virtual environment, including the VMM/Host OS, and all VMs. This will operate on real hardware and save real power. The main guideline for global power management is that only the owner can do real power management operation to that device. And VMM/Host OS is responsible for coordinating the power management sequence.

This paper will elaborate how power management state (*Sx/Cx/Px/Dx*) is implemented for both virtual and real types.

2 Sleep States (*Sx*) Support

Linux currently supports *S1* (stand-by), *S3* (suspend-to-ram) and *S4* (suspend-to-disk). This section mainly discusses the *S3* and *S4* state support in the virtualization environment. *S1* and *S3* are similar, so the *S3* discussion can also apply to *S1*.

2.1 Virtual *S3*

Virtual *S3* is *S3* suspend/resume within a virtual machine, which is similar to native. When guest OSes see that the virtual platform has *S3* capability, it can start *S3* process either requested by user or forced by control tool under certain predefined condition (e.g. VM being idle for more than one hour). Firstly, the Guest OS freezes all processes and also write a wakeup vector to virtual ACPI FACS table. Then, the Guest OS saves all contexts, including I/O device context and CPU context. Finally, the Guest OS will issue hardware *S3* command, which is normally I/O port writing. VMM will capture the I/O port writing and handle the *S3* command by resetting the virtual CPU. The VM is now in virtual sleep state.

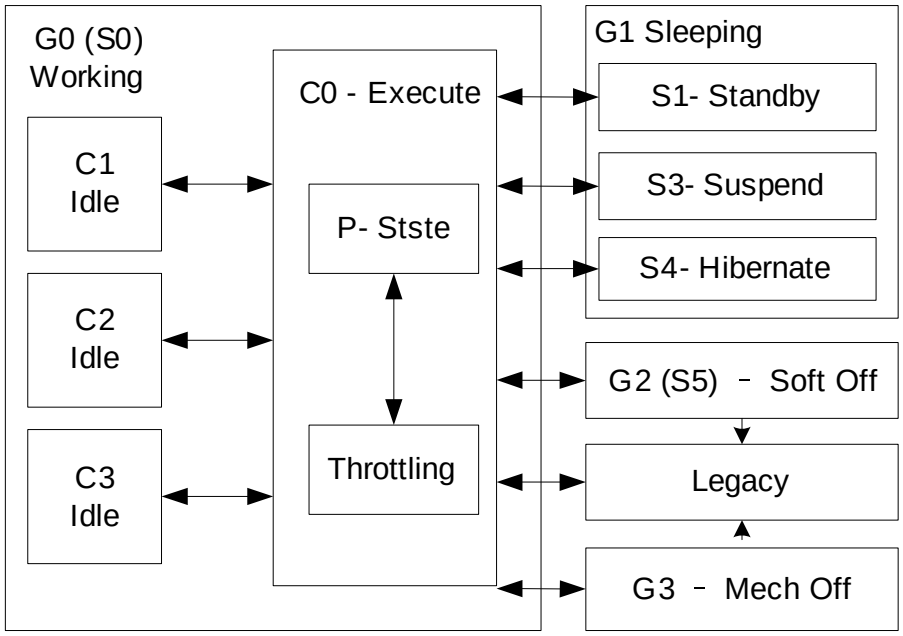


Figure 1: ACPI State Relationship

Guest OS *S3* wakeup is a reverse process. Firstly, VMM will put the virtual CPU into real mode, and start virtual CPU from guest BIOS POST code. BIOS POST will detect that it is a *S3* wakeup and thus jump to the *S3* wakeup vector stored in guest ACPI FACS table. The wakeup routine in turn will restore all CPU and I/O context and unfreeze all processes. Now the Guest OS resumes to working state.

From the above virtual *S3* suspend/resume process, it is easy to see that VMM needs the following work to support virtual *S3*:

- **Guest ACPI Table**: the ACPI DSDT table should have *_S3* package to tell guest OS that the virtual platform has *S3* capability, otherwise, guest OS won't even start *S3* sleep. Guest ACPI table can also have optional OEM-specific fields if required.

- **Guest BIOS POST Code**: Logic must be added here to detect the *S3* resume and get wakeup vector address from ACPI FACS table, and then jump to wakeup vector.

- ***S3* Command Interception**: Firstly, device model should emulate the ACPI PM1A control register, so that it can capture the S3 request. In KVM and Xen case, this can be done in QEMU side, and is normally implemented as a system I/O port. Secondly, to handle S3 request, VMM need to reset all virtual CPUs.

2.2 Real *S3* State

Unlike virtual *S3*, Real *S3* will put the whole system into sleep state, including VMM/Host OS and all the virtual machines. So it is more meaningful in terms of power saving.

Linux already has fundamental *S3* support, like to freeze/unfreeze processes, suspend/resume I/O devices, hotplug/unplug CPUs for SMP case, etc. to conduct a complete S3 suspend/resume process.

Real *S3* in virtualization also need similar sequence as above. The key difference is that system resources may be owned by different component. So the guideline is to ensure right owner to suspend/resume its owned resource.

Take Xen as an example. The suspend/resume operation must be coordinated among hypervisor, privileged VM and driver domain. Most I/O devices are owned by a privileged VM (domain0) and driver domain, so suspend/resume on those devices is mostly done in domain0 and driver domain. Then hypervisor will cover the rest:

- **Hypervisor owned devices**: APIC, PIC, UART, platform timers like PIT, etc. Hypervisor needs to suspend/resume those devices
- **CPU**: owned by hypervisor, and thus managed here
- **Wakeup routine**: At wakeup, hypervisor need to be the first one to get control, so wakeup routine is also provided by hypervisor.
- **ACPI PM1x control register**: Major ACPI sleep logic is covered by domain0 with the only exception of PM1x control register. Domain0 will notify hypervisor at the place where it normally writes to PM1x register. Then hypervisor covers the above work and write to this register at the final stage, which means a physical *S3* sleep.

For the driver domain that is assigned with physically I/O device, hypervisor will notify these domains to do virtual *S3* first, so that these domains will power off their I/O device before domain0 starts its sleep sequence.

Figure 2 illustrates the Xen Real *S3* sequence.

2.3 Virtual *S4* State and Real *S4* State

Virtual *S4* is suspend-to-disk within virtual machine. Guest OS is responsible to save all contexts (CPU, I/O device, memory) to disk and enter sleep state. Virtual *S4* is a useful feature because it can reduce guest OS booting time.

From the VMM point of view, virtual *S4* support implementation is similar as virtual *S3*. The guest ACPI also needs to export *S4* capability and VMM needs to capture the *S4* request. The major difference is how VMM handles the *S3/S4* request. In *S3*, VMM needs resetting VCPU in *S3* and jumps to wakeup vector when VM resuming. In *S4*, VMM only needs to destroy the VM since VMM doesn't need to preserve the VM memory. To resume from *S4*, user can recreate the VM with the previous disk image, the guest OS will know that it is *S4* resume and start resuming from *S4*.

Real *S4* state support is also similar as native *S4* state. For host-based model, it can leverage host OS *S4* support directly. But it's more complex in a hybrid model like Xen. The key design issue is how to coordinate hypervisor and domain0 along the suspend process. For example, disk driver can be only suspended after VMM dumps its own memory into disk. Then once hypervisor finishes its memory dump, later change on virtual CPU context of domain0 can not be saved any more. After wakeup, both domain0 and hypervisor memory image need to be restored and sequence is important here. This is still an open question.

3 Processor Power States (*Cx*) support

Processor power states, while in the *G0* working state, generally refer to active or idle state on the CPU. *C0* stands for a normal power state where CPU dispatches and executes instructions, and *C1*, *C2* $\cdots$ *Cn* indicates low-power idle states where no instructions are executed and power consumption is reduced to a different level. Generally speaking, a larger value of *Cx* brings greater power savings, at the same time adds longer entry/exit latency. It's important for OSPM to understand ability and implication of each C-state, and then define appropriate policy to suit activities of the time:

- Methods to trigger specific C-state
- Worst case latency to enter/exit C-state
- Average power consumption at given C-state

Progressive policy may hurt some components which don't tolerate big delay, while conservative policy makes less use of power-saving capability provided by hardware. For example, OSPM should be aware that cache coherency is not maintained by the processor when in *C3* state, and thus needs to manually flush cache before entering when in SMP environment. Based on different hardware implementation, TSC may be stopped and so does LAPIC timer interrupt. When *Cx* comes into virtualization, things become more interesting.

3.1 Virtual C-states

Virtual C-states are presented to VM as a 'virtual' power capability on 'virtual' processor. The straight-forward effect of virtual C-states is to exit virtual processor from scheduler when *Cx* is entered, and to wake virtual processor back to scheduler upon break event. Since virtual processor is 'virtual' context created and managed by VMM, transition among virtual C-states have nothing

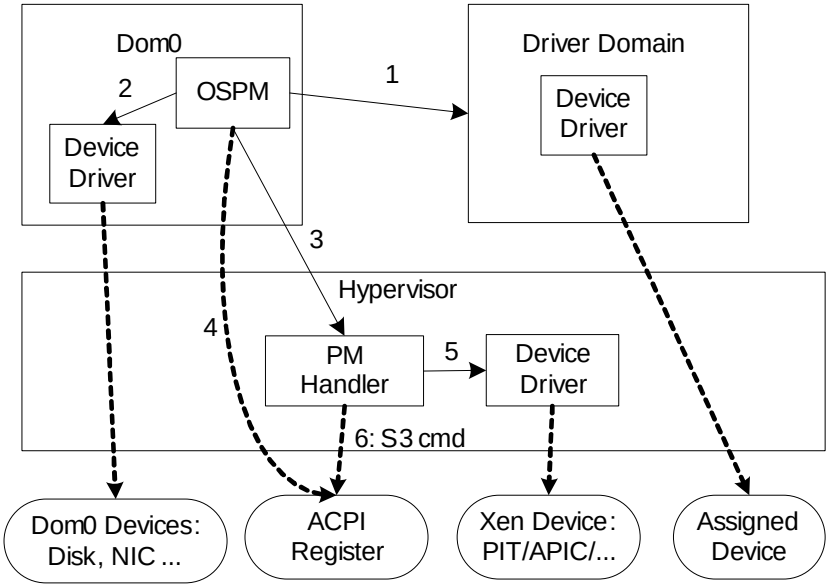


Figure 2: Xen Real *S3* sequence

to do with power state on real processors, but does have the ability to provide useful hints in some cases.

The way to implement virtual C-states can vary upon the virtualization model. For example, a hardware-assisted guest may be presented with C-states capability fully conforming to ACPI specification, while a para-virtualized guest can simply take quick hyper-call to request. Basically it doesn't make much sense to differentiate among *C1*, *C2*$\cdots$*Cn* in a virtualization world, but it may be useful to some cases. One direct case is to test processor power management logic of a given OSPM, or even to try whether some newer C-state is a meaningful model before hardware is ready. Another interesting usage would be to help physical C-state governor for right decision, since virtual C-state request reveals activities within a guest.

3.1.1 Para-virtualized guest

para-virtualized guest is a modified guest which can cooprate with VMM to improve performance. virtual C-state for a para-virtualized guest just borrows the term from ACPI, but no need to bind with any ACPI context. A simple policy can just provide 'active' and 'sleep' categories for a virtual processor, without differentiation about *C1*$\cdots$*Cn*. When idle thread is scheduled without anything to handle, time events in the near future are walked for nearest interval which is then taken as parameter of sleep hyper-call issued to VMM. Then VMM drops the virtual CPU from the run-queue and may wake it up later upon any break event (like interrupt) or specified interval timeout. A perfect match on a tick-less time model! Since it's more like the normal 'HALT' instruction usage, the policy is simple which is tightly coupled with time sub-system.

It's also easy to extend para-virtualized guest with more fine-grained processor power states, by extending above hyper-calls. Such hyper-call based interface can be hooked into generic Linux processor power management infrastructure, with common policies unchanged but a different low-level power control interface added.

3.1.2 Hardware-assisted guest

Hardware-assisted guest is the unmodified guest with hardware (e.g. Intel VT or AMD-V) support. Not like a para-virtualized guest who poses changes within the guest, virtual platform (i.e., device model) needs export exact control interface conforming to ACPI spec and emulate desired effect as what hardware-assisted guest expects. By providing the same processor power management capability, no change is required within hardware-assisted guest.

- **Virtual *C2*** – an ability provided by chipset which

controls clock input signal. First device model needs to construct correct ACPI table to expose related information, including trigger approach, latency and power consumption as what ACPI spec defines. ACPI FADT table contains fixed format information, like P_LVL2 command register for trigger. Recent ACPI version also adds a more generic object _CST to describe all C-state information, e.g. C state dependency and mwait extension. Device model may want to provide both methods if taken as a test environment.

Device model then needs to send a notification to VMM after detecting virtual C2 request from guest. As acceleration, Cx information can be registered to VMM and then VMM can handle directly. Actually, for virtual C2 state, device model doesn't need to be involved at run time. C2 is defined as a low-power idle state with bus snooped and cache coherency maintained. Basic virtual MMU management and DMA emulation have already ensured this effect at given time.

- **Virtual *C3*** – almost the same as virtual *C2*, P_LVL3 or _CST describe the basic information. But virtual *C3* also affects device model besides virtual processor. Device model needs to provide PM2_CNT.ARB_DIS which disables bus master cycles and thus DMA activities. PM1x_STS.BM_STS, an optional feature of chipset virtualization, reveals bus activity status which is a good hint for OSPM to choose *C2* or *C3*. More importantly, PM1x_CNT.BM_RLD provides option to take bus master requests as break event to exit *C3*. To provide correct emulation, tight cooperation between device model and VMM is required which brings overhead. So it's reasonable for device model to give up such support, if not aimed to test OSPM behavior under *C3*.

- **Deeper virtual *Cx*** – similar as *C3*, and more chipset logic virtualization are required.

3.2 Real C-states

VMM takes ownership of physical CPUs and thus is required to provide physical C-states management for 'real' power saving. The way to retrieve C-states information and conduct transition is similar to what today OSPM does according to ACPI spec. For a host based VMM like KVM, those control logic has been there in the host environment and nothing needs to be changed. Then, for a hybrid VMM like Xen, domain0 can parse and register C-state information to hypervisor which is equipped with necessary low-level control infrastructure.

There are some interesting implementation approaches . For example, VMM can take a virtual *Cx* request into consideration. Normally guest activities occupy major portion of cpu cycles which can then be taken as a useful factor for C-state decision. VMM may then track the virtual C-state requests from different guests, which represent the real activities on given CPU. That info can be hooked into existing governors to help make better decisions. For example:

- Never issue a C-x transition if no guest has such virtual C-x request pending
- Only issue a C-x transition only if all guests have same virtual C-x requests
- Pick the C-x with most virtual C-x requests in the given period

Of course, the above is very rough and may not result in a really efficient power saving model. For example, one guest with poor C-state support may prevent the whole system from entering a deeper state even when condition satisfies. But it does be a good area for research to leverage guest policies since different OS may have different policy for its specific workload.

4 Processor Performance States (*Px*) Support

P-states provide OSPM an opportunity to change both frequency and voltage on a given processor at run-time, which thus brings more efficient power-saving ability. Current Linux has several sets of governors, which can be user-cooperative, static, or on-demand style. P-states within the virtualization world are basically similar to the above C-states discussion in many concepts, and thus only its specialties are described below.

4.1 Virtual P-states

Frequency change on a real processor has the net effect to slow the execution flow, while voltage change is at

fundamental level to lower power consumption. When coming to virtual processor, voltage is a no-op but frequency does have useful implication to the scheduler. Penalty from scheduler has similar slow effect as frequency change. Actually we can easily plug virtual P-state requests into schedule information, for example:

- Half its weight in a weight-based scheduler
- Lower its priority in a priority-based scheduler

Furthermore, scheduler may bind penalty level to different virtual P-state, and export this information to guest via virtual platform. Virtual platform may take this info and construct exact P-states to be presented to guest. For example, *P1* and *P2* can be presented if scheduler has two penalty levels defined. These setup the bridge between virtual P-states and scheduler hints. Based on this infrastructure, VMM is aware of guest requirement and then grant cycles more efficiently to guest with more urgent workload.

4.2 Real P-states

Similar as real C-states for virtualization, we can either reuse native policy or add virtualization hints. One interesting extension is based on user space governor. We can connect together all guest user space governors and have one governor act as the server to collect that information. This server can be a user space governor in host for a host-based VMM, or in privileged VM for hybrid VMM. Then, this user space governor can incorporate decisions from other user space governors and then make a final one. Another good point for this approach is that hybrid VMM can directly follow request from privileged VM by a simple "follow" policy.

5 Device Power States (*Dx*) Support

Devices consume another major portion of power supply, and thus power feature on devices also plays an important role. Some buses, like PCI, have well-defined power management feature for devices, and ACPI covers the rest if missing. Power state transition for a given device can be triggered in either a passive or active way. When OSPM conducts a system level power state transition, like *S3*/*S4*, all devices are forced to enter appropriate low power state. OSPM can also introduce active on-demand device power management at run-time, on some device if inactive for some period. Carefulness must be taken to ensure power state change of one node does not affect others with dependency. For example, all the nodes on a waken path have to satisfy minimal power requirement of that wake method.

5.1 Virtual D-states

Devices seen by a guest are basically split into three categories: emulated, para-virtualized, and direct-assigned. Direct-assigned devices are real with nothing different regarding D-states. Emulated and para-virtualized are physically absent, and thus device power states on them are also virtual.

Normally, real device class defines what subset of capabilities are available at each power level. Then, by choosing the appropriate power state matching functional requirement at the time, OSPM can request device switching to that state for direct power saving at the electrical level. Virtual devices, instead, are completely software logics either emulated as a real device or para-virtualized as a new device type. So virtual D-states normally show as reduction of workload, which has indirect effect on processor power consumption and thus also contributes to power saving.

For emulated devices, the device model presents exact same logic and thus D-states definition as a real one. Para-virtualized devices normally consist of front-end and back-end drivers, and connection states between the pair can represent the virtual D-states. Both device model and back-end need to dispatch requests from guest, and then handle with desired result back. Timer, callback, and kernel thread, etc. are possible components to make such process efficient. As a result of virtual D-states change, such resources may be frozen or even freed to reduce workload imposed on the physical processor. For example, the front-end driver may change connection state to 'disconnected' when OSPM in guest requests a D3 state transition. Then, back-end driver can stop the dispatch thread to avoid any unnecessary activity caused in the idle phase. Same policy also applies to device model which may, for example, stop timer for periodically screen update.

Virtual bus power state can be treated with same policy as virtual device power state, and in most time may be just a no-op if virtual bus only consists of function calls.

5.2 Real D-states

Real device power states management in virtualization case are a bit complex, especially when device may be direct assigned to guests (known as a driver domain). To make this area clear, we first show the case without driver domain, and then unveil tricky issues when the later is concerned.

5.2.1 Basic virtualization environment

Basic virtualization model have all physical devices owned by one privileged component, say host Linux for KVM and domain-0 for Xen. OSPM of that privileged guy deploys policies and takes control of device power state transitions. Device model or back-end driver are clients on top of related physical devices, and their requests are counted into OSPM's statistics for given device automatically. So there's nothing different to existing OSPM.

For example, OSPM may not place disk into deeper D-states when device model or back-end driver is still busy handling disk requests from guest which adds to the workload on real disk.

As comparison to the OSPM within guests, we refer to this special OSPM as the "dominate OSPM." Also dominator is alias to above host Linux and domain-0 in below context for clear.

5.2.2 Driver domains

Driver domains are guests with some real devices assigned exclusively, to either balance the I/O virtualization bottleneck or simply speed the guest directly. The fact that OSPM needs to care about the device dependency causes a mismatch on this model: dominate OSPM with local knowledge needs to cover device dependencies across multiple running environments.

A simple case (Figure 3) is to assign *P2* under PCI Bridge1 to guest GA, with the rest still owned by dominator. Say an on-demand D-states governor is active in the dominate OSPM, and all devices under *Bridge1* except *P2* have been placed into *D3*. Since all the devices on bus 1 are inactive now based on local knowledge, dominate OSPM may further decide to lower power

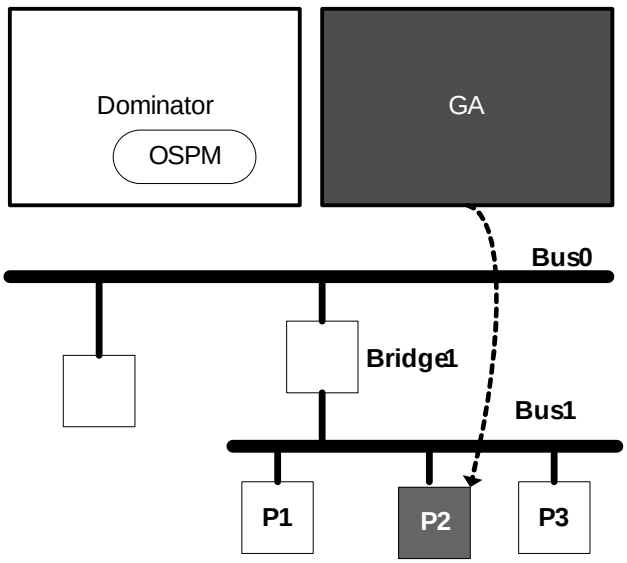


Figure 3: A simple case

voltage and stop clock on bus 1 by conducting *Bridge1* into a deeper power state. Devil take it! *P2* can never work now, and GA has to deal with a dead device without response.

Then, the idea is simple to kick this issue: extend local dominate OSPM to construct full device tree across all domains. The implication is that on-demand device power governor can't simply depend on in-kernel statistics, and hook should be allowed from other components. Figure 4 is one example of such extension:

Device assignment means grant of port I/O, MMIO, and interrupt in substance, but the way to find assigned device is actually virtualized. For example, PCI device discovery is done by PCI configuration space access, which is virtualized in all cases as part of virtual platform. That's the trick of how the above infrastructure works. For hardware-assisted guest, device model intercepts access by traditional 0xcf8/0xcfc or memory mapped style. Para-virtualized guest can have a PCI frontend/backend pair to abstract PCI configuration space operation, like already provided by today's Xen. Based on this reality, device model or PCI backend can be good place to reveal device activity if owned by other guests, since standard power state transition is done by PCI configuration space access as defined by PCI spec. Then based on hint from both in-kernel and other virtualization related components, dominate OSPM can now precisely decide when to idle a parent node if with child nodes shared among guests.

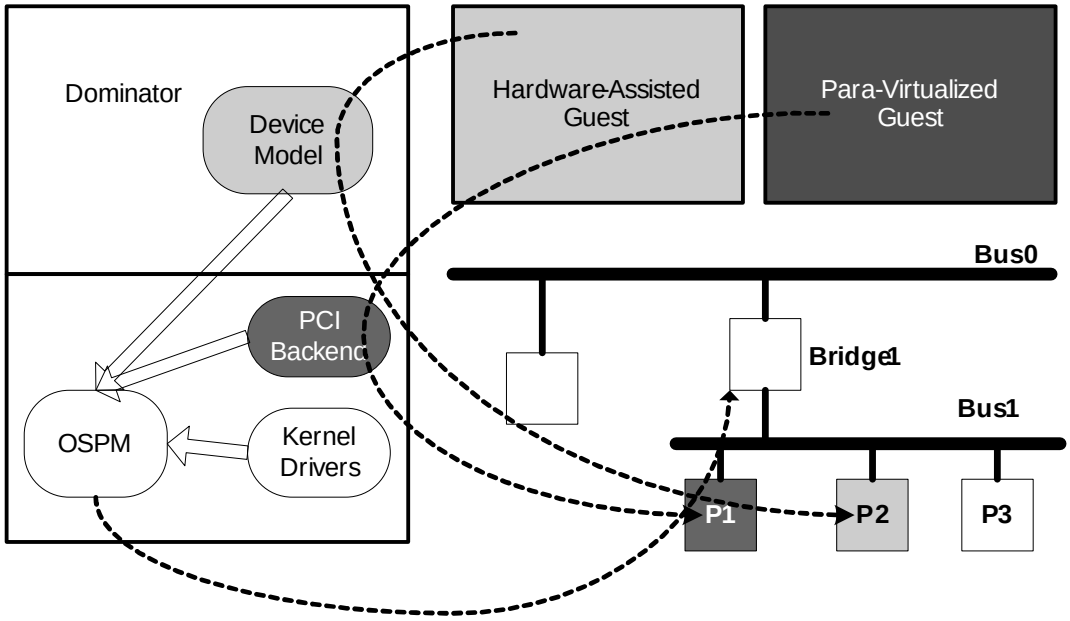


Figure 4: The extended case

However, when another bus type is concerned without explicit power definition, it's more complex to handle device dependency. For example, for devices with power information provided by ACPI, the control method is completely encapsulated within ACPI AML code. Then the way to intercept power state change has to be a case specific approach, based on ACPI internal knowledge. Fortunately, most of the time only PCI devices are preferred regarding the device assignment.

6 Current Status

Right now our work on this area is mainly carried on Xen. Virtual *S3*/*S4* to hardware-assisted guest has been supported with some extension to ACPI component within QEMU. This should also apply to other VMM software with same hardware-assisted support.

Real *S3* support is also ready. Real S3 stability relies on the quality of Linux S3 support, since domain0 as a Linux takes most responsibility with the only exception at final trigger point. Some linux S3 issues are met. For example, SATA driver with AHCI mode has stability issue on 2.6.18 which unfortunately is the domain0 kernel version at the time. Another example is the VGA resume. Ideally, real systems that support Linux should restore video in the BIOS. Real native Linux graphics drivers should also restore video when they are used. If it does not work, you can find some workaround in `documentation/power/video.txt`. The positive side is that Linux S3 support is more and more stable as time goes by. Real S4 support has not been started yet.

Both virtual and real Cx/Px/Tx/Dx supports are in development, which are areas with many possibilities worthy of investigation. Efficient power management policies covering both virtual and real activities are very important to power saving in a run-time virtualization environment. Forenamed sections are some early findings along with this investigation, and surely we can anticipate more fun from this area in the future.

References

[1] "Advanced Configuration & Power Specification," Revision 3.0b, 2006, Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba.
`http://www.acpi.info`

[2] "ACPI in Linux," L. Brown, A. Keshavamurthy, S. Li, R. Moore, V. Pallipadi, L. Yu, In *Proceedings of the Linux Symposium* (OLS), 2005.

[3] "Xen 3.0 and the Art of Virtualization," I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D.

Magenheimer, J. Nakajima, and A. Mallick, in *Proceedings of the Linux Symposium* (OLS), 2005.

Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps

Jim Keniston
IBM
jkenisto@us.ibm.com

Ananth Mavinakayanahalli
IBM
ananth@in.ibm.com

Prasanna Panchamukhi
IBM
prasanna@in.ibm.com

Vara Prasad
IBM
varap@us.ibm.com

Abstract

The ptrace system-call API, though useful for many tools such as gdb and strace, generally proves unsatisfactory when tracing multithreaded or multi-process applications, especially in timing-dependent debugging scenarios. With the utrace kernel API, a kernel-side instrumentation module can track interesting events in traced processes. The uprobes kernel API exploits and extends utrace to provide kprobes-like, breakpoint-based probing of user applications.

We describe how utrace, uprobes, and kprobes together provide an instrumentation facility that overcomes some limitations of ptrace. For attendees, familiarity with a tracing API such as ptrace or kprobes will be helpful but not essential.

1 Introduction

For a long time now, debugging user-space applications has been dependent on the ptrace system call. Though ptrace has been very useful and will almost certainly continue to prove its worth, some of the requirements it imposes on its clients are considered limiting. One important limitation is performance, which is influenced by the high context-switch overheads inherent in the ptrace approach.

The utrace patchset [3] mitigates this to a large extent. Utrace provides in-kernel callbacks for the same sorts of events reported by ptrace. The utrace patchset reimplements ptrace as a client of utrace.

Uprobes is another utrace client. Analogous to kprobes for the Linux® kernel, uprobes provides a simple, easy-to-use API to dynamically instrument user applications. Details of the design and implementation of uprobes form the major portion of this paper.

We start by discussing the current situation in the user-space tracing world. Sections 2 and 3 discuss the various instrumentation approaches possible and/or available. Section 4 goes on to discuss the goals that led to the current uprobes design, while Section 5 details the implementation. In the later sections, we put forth some of the challenges, especially with regard to modifying text and handling of multithreaded applications. Further, there is a brief discussion on how and where we envision this infrastructure can be put to use. We finally conclude with a discussion on where this work is headed.

2 Ptrace-based Application Tracing

Like many other flavors of UNIX, Linux provides the ptrace system-call interface for tracing a running process. This interface was designed mainly for debugging, but it has been used for tracing purposes as well. This section surveys some of the ptrace-based tracing tools and presents limitations of the ptrace approach for low-impact tracing.

Ptrace supports the following types of requests:

- Attach to, or detach from, the process being traced (the “tracee”).
- Read or write the process’s memory, saved registers, or user area.
- Continue execution of the process, possibly until a particular type of event occurs (e.g., a system call is called or returns).

Events in the tracee turn into `SIGCHLD` signals that are delivered to the tracing process. The associated `siginfo_t` specifies the type of event.

2.1 Gdb

gdb is the most widely used application debugger in Linux, and it runs on other flavors of UNIX as well. *gdb* controls the program to be debugged using ptrace requests. *gdb* is used mostly as an interactive debugger, but also provides a batch option through which a series of *gdb* commands can be executed, without user intervention, each time a breakpoint is hit. This method of tracing has significant performance overhead. *gdb*'s approach to tracing multithreaded applications is to stop all threads whenever any thread hits a breakpoint.

2.2 Strace

The *strace* command provides the ability to trace calls and returns from all the system calls executed by the traced process. *strace* exploits ptrace's `PTRACE_SYSCALL` request, which directs ptrace to continue execution of the tracee until the next entry or exit from a system call. *strace* handles multithreaded applications well, and it has significantly less performance overhead than the *gdb* scripting method; but performance is still the number-one complaint about *strace*. Using *strace* to trace itself shows that each system call in the tracee yields several system calls in *strace*.

2.3 Ltrace

The *ltrace* command is similar to *strace*, but it traces calls and returns from dynamic library functions. It can also trace system calls, and extern functions in the traced program itself. *ltrace* uses ptrace to place breakpoints at the entry point and return address of each probed function. *ltrace* is a useful tool, but it suffers from the performance limitations inherent in ptrace-based tools. It also appears not to work for multithreaded programs.

2.4 Ptrace Limitations

If *gdb*, *strace*, and *ltrace* don't give you the type of information you're looking for, you might consider writing your own ptrace-based tracing tool. But consider the following ptrace limitations first:

- Ptrace is not a POSIX system call. Its behavior varies from operating system to operating system, and has even varied from version to version in Linux. Vital operational details ("Why do I get two SIGCHLDs here? Am I supposed to pass the process a SIGCONT or no signal at all here?") are not documented, and are not easily gleaned from the kernel source.
- The amount of perseverence and/or luck you need to get a working program goes up as you try to monitor more than one process or more than one thread.
- Overheads associated with accessing the tracee's memory and registers are enormous—on the order of 10x to 100x or more, compared with equivalent in-kernel access. Ptrace's PEEK-and-POKE interface provides very low bandwidth and incurs numerous context switches.
- In order to trace a process, the tracer must become the tracee's parent. To attach to an already running process, then, the tracer must muck with the tracee's lineage. Also, if you decide you want to apply more instrumentation to the same process, you have to detach the tracer already in place.

3 Kernel-based Tracing

In the early days of Linux, the kernel code base was manageable and most people working on the kernel knew their core areas intimately. There was a definite pushback from the kernel community towards including any tracing and/or debugging features in the mainline kernel.

Over time, Linux became more popular and the number of kernel contributors increased. A need for a flexible tracing infrastructure was recognized. To that end, a number of projects sprung up and have achieved varied degrees of success.

We will look at a few of these projects in this section. Most of them are based on the kernel-module approach.

3.1 Kernel-module approach

The common thread among the following approaches is that the instrumentation code needs to run *in kernel*

mode. Since we wouldn't want to burden the kernel at times when the instrumentation isn't in use, such code is introduced only when needed, in the form of a kernel module.

3.1.1 Kprobes

Kprobes [2] is perhaps the most widely accepted of all dynamic instrumentation mechanisms currently available for the Linux kernel. Kprobes traces its roots back to DProbes (discussed later). In fact, the first version of kprobes was a patch created by just taking the minimal, kernel-only portions of the DProbes framework.

Kprobes allows a user to dynamically insert "probes" into specific locations in the Linux kernel. The user specifies "handlers" (instrumentation functions) that run before and/or after the probed instruction is executed. When a probepoint (which typically is an architecture-specific breakpoint instruction) is hit, control is passed to the kprobes infrastructure, which takes care of executing the user-specified handlers. [2] provides an in-depth treatment of the kprobes infrastructure (which, incidentally, includes jprobes and function-return probes).

The kernel-module approach was a natural choice for kprobes: after all, the goal is to access and instrument the Linux kernel. Given the privilege and safety requirements necessary to access kernel data structures, the kernel-module approach works very well.

3.1.2 Utrace

A relatively new entrant to this instrumentation space is utrace. This infrastructure is intended to serve as an abstraction layer to write the next generation of user-space tracing and debugging applications.

One of the primary grouses kernel hackers have had about ptrace is the lack of separation/layering of code between architecture-specific and -agnostic parts. Utrace aims to mitigate this situation. Ptrace is now but one of the clients of utrace.

Utrace, at a very basic level, is an infrastructure to monitor individual Linux "threads"—each represented by a `task_struct` in the kernel. An "engine" is utrace's basic control unit. Typically, each utrace client establishes an engine for each thread of interest. Utrace provides three basic facilities on a per-engine basis:

- *Event reporting:* Utrace clients register callbacks to be run when the thread encounters specific events of interest. These include system call entry/exit, signals, exec, clone, exit, etc.
- *Thread control:* Utrace clients can inject signals, request that a thread be stopped from running in user-space, single-step, block-step, etc.
- *Thread machine state access:* While in a callback, a client can inspect and/or modify the thread's core state, including the registers and u-area.

Utrace works by placing tracepoints at strategic points in kernel code. For traced threads, these tracepoints yield calls into the registered utrace clients. These callbacks, though happening in the context of a user process, happen when the process is executing in kernel mode. In other words, utrace clients run in the kernel. Ptrace is one utrace client that lives in the kernel. Other clients—especially those used for *ad hoc* instrumentation—may be implemented as kernel modules.

3.1.3 Uprobes

Uprobes is another client of utrace. As such, uprobes can be seen as a flexible user-space probing mechanism that comes with all the power, but not all the constraints, of ptrace. Just as kprobes creates and manages probepoints in kernel code, uprobes creates and manages probepoints in user applications. The uprobes infrastructure, like ptrace, lives in the kernel.

A uprobes user writes a kernel module, specifying for each desired probepoint the process and virtual address to be probed and the handler to run when the probepoint is hit.

A uprobes-based module can also use the utrace and/or kprobes APIs. Thus a single instrumentation module can collect and correlate information from the user application(s), shared libraries, the kernel/user interfaces, and the kernel itself.

3.1.4 SystemTap

SystemTap [4] provides a mechanism to use the kernel (and later, user) instrumentation tools, through a simple

awk-like event/action scripting language. A SystemTap script specifies code points to probe, and for each, the instrumentation code to run when the probepoint is hit. The scripting language provides facilities to aggregate, collate, filter, present, and analyze trace data in a manner that is intuitive to the user.

From a user's script, the *stap* command produces and loads a kernel module containing calls to the kprobes API. Trace data from this module is passed to user space using efficient mechanisms (such as relay channels), and the SystemTap post-processing infrastructure takes care of deciphering and presenting the gathered information in the format requested by the user, on the *stap* command's standard output.

SystemTap can be viewed as a wrapper that enables easy use of kernel instrumentation mechanisms, including, but not limited to, kprobes. Plans are afoot to exploit utrace, uprobes, the "markers" static tracing infrastructure, and a performance-monitoring-hardware infrastructure (perfmon), as they become part of the mainline kernel.

3.2 Interpreter-based approaches

Two more instrumentation tools, DProbes and DTrace, bear mention. Both tools endeavor to provide integrated tracing support for user applications and kernel code.

DProbes [1] traces its roots to IBM's OS/2® operating system, but has never found a foothold in Linux, except as the forebear of kprobes. DProbes instrumentation is written in the RPN programming language, the assembly language for a virtual machine whose interpreter resides in the kernel. For the i386 architecture, the DProbes C Compiler (dpcc) translates instrumentation in a C-like language into RPN. Like kprobes and uprobes, DProbes allows the insertion of probepoints anywhere in user or kernel code.

DTrace is in use on Solaris and FreeBSD. Instrumentation is written in the D programming language, which provides aggregation and presentation facilities similar to those of SystemTap. As with DProbes, DTrace instrumentation runs on an interpreter in the kernel. DTrace kernel probepoints are limited to a large but predefined set. A DTrace probe handler is limited to a single basic block (no ifs or loops).

Tool	Event Counted	Overhead (usec) per Event
ltrace -c	function calls	22
gdb -batch	function calls	265
strace -c	system calls	25
kprobes	function calls	0.25
uprobes	function calls	3.4
utrace	system calls	0.16

Table 1: Performance of ptrace-based tools (top) vs. *ad hoc* kernel modules (bottom)

3.3 Advantages of kernel-based application tracing

- Kernel-based instrumentation is inherently fast. Table 1 shows comparative performance numbers of ptrace-based tools vs. kernel modules using kprobes, uprobes, and utrace. These measurements were taken on a Pentium® M (1495 MHz) running the utrace implementation of ptrace.
- Most systemic problems faced by field engineers involve numerous components. Problems can percolate from a user-space application all the way up to a core kernel component, such as the block layer or the networking stack. Providing a unified view of the flow of control and/or data across user space and kernel is possible only via kernel-based tracing.
- Kernel code runs at the highest privilege and as such, can access all kernel data structures. In addition, the kernel has complete freedom and access to the process address space of all user-space processes. By using safe mechanisms provided in kernel to access process address spaces, information related to the application's data can be gleaned easily.
- Ptrace, long the standard method for user-space instrumentation, has a number of shortcomings, as discussed in Section 2.4.

3.4 Drawbacks of kernel-based application tracing

- The kernel runs at a higher privilege level and in a more restricted environment than applications. Assumptions and programming constructs that are valid for user space don't necessarily hold for the kernel.

- Not everybody is a kernel developer. It's not prudent to expect someone to always write "correct" kernel code. And, while dealing with the kernel, one mistake may be too many. Most application developers and system administrators, understandably, are hesitant to write kernel modules.
- The kernel has access to user-space data, but can't easily get at all the information (e.g., symbol table, debug information) to decode it. This information must be provided in or communicated to the kernel module, or the kernel must rely on post-processing in user space.

SystemTap goes a long way in mitigating these drawbacks. For kernel tracing, SystemTap uses the running kernel's debug information to determine source file, line number, location of variables, and the like. For applications that adhere to the DWARF format, it wouldn't be hard for SystemTap to provide address/symbol resolution.

4 Uprobes Design Goals

The idea of user-space probes has been around for years, and in fact there have been a variety of proposed implementations. For the current, utrace-based implementation, the design goals were as follows:

- *Support the kernel-module approach.* Uprobes follows the kprobes model of dynamic instrumentation: the uprobes user creates an *ad hoc* kernel module that defines the processes to be probed, the probepoints to establish, and the kernel-mode instrumentation handlers to run when probepoints are hit. The module's `init` function establishes the probes, and its cleanup function removes them.
- *Interoperate with kprobes and utrace.* An instrumentation module can establish probepoints in the kernel (via kprobes) as well as in user-mode programs. A uprobe or utrace handler can dynamically establish or remove kprobes, uprobes, or utrace-event callbacks.
- *Minimize limitations* on the types of applications that can be probed and the way in which they can be probed. In particular, a uprobes-based instrumentation module can probe any number of processes of any type (related or unrelated), and can probe multithreaded applications of all sorts. Probes can be established or removed at any stage in a process's lifetime. Multiple instrumentation modules can probe the same processes (and even the same probepoints) simultaneously.
- *Probe processes, not executables.* In earlier versions of uprobes, a probepoint referred to a particular instruction in a specified executable or shared library. Thus a probepoint affected all processes (current and future) running that executable. This made it relatively easy to probe a process right from exec time, but the implications of this implementation (e.g., inconsistency between the in-memory and on-disk images of a probed page) were unacceptable. Like ptrace, uprobes now probes per-process, and uses `access_process_vm()` to ensure that a probed process gets its own copy of the probed page when a probepoint is inserted.
- *Handlers can sleep.* As discussed later in Section 5, uprobes learns of each breakpoint hit via utrace's signal-callback mechanism. As a result, the user-specified handler runs in a context where it can safely sleep. Thus, the handler can access any part of the probed process's address space, resident or not, and can undertake other operations (such as registering and unregistering probes) that a kprobe handler cannot. Compared with earlier implementations of uprobes, this provides more flexibility at the cost of some additional per-hit overhead.
- *Performance.* Since a uprobe handler runs in the context of the probed process, ptrace's context switches between the probed process and the instrumentation parent on every event are eliminated. The result is significantly less overhead per probe hit than in ptrace, though significantly more than in kprobes.
- *Safe from user-mode interference.* Both uprobes and the instrumentation author must account for the possibility of unexpected tinkering with the probed process's memory—e.g., by the probed process itself or by a malicious ptrace-based program—and ensure that while the sabotaged process may crash, the kernel's operation is not affected.
- *Minimize intrusion on existing Linux code.* For hooks into a process's events of interest, uprobes

uses those provided by utrace. Uprobes creates per-task and per-process data structures, but maintains them independently of the corresponding data structures in the core kernel and in utrace. As a result, at this writing, the base uprobes patch, including i386 support, includes only a few lines of patches against the mainline kernel source.

- *Portable to multiple architectures.* At the time of this writing, uprobes runs on the i386 architecture. Ports are underway to PowerPC, x86_64, and zLinux (s390x). Except for the code for single-stepping out of line, which you need (adapting from kprobes, typically) if you don't want to miss probepoints in multithreaded applications, a typical uprobes port is on the order of 50 lines.

5 Uprobes Implementation Overview

Uprobes can be thought of as containing the following overlapping pieces:

- data structures;
- the register/unregister API;
- utrace callbacks; and
- architecture-specific code.

In this section, we'll describe each of these pieces briefly.

5.1 Data structures

Uprobes creates the following internal data structures:

- `uprobe_process` – one for each probed process;
- `uprobe_task` – one for each thread (task) in a probed process; and
- `uprobe_kimg` (Kernel IMaGe) – one for each probepoint in a probed process. (Multiple uprobes at the same address map to the same `uprobe_kimg`.)

Each `uprobe_task` and `uprobe_kimg` is owned by its `uprobe_process`. Data structures associated with return probes (uretprobes) and single-stepping out of line are described later.

5.2 The register/unregister API

The fundamental API functions are `register_uprobe()` and `unregister_uprobe()`, each of which takes a pointer to a uprobe object defined in the instrumentation module. A uprobe object specifies the pid and virtual address of the probepoint, and the handler function to be run when the probepoint is hit.

The `register_uprobe()` function finds the `uprobe_process` specified by the pid, or creates the `uprobe_process` and `uprobe_task(s)` if they don't already exist. `register_uprobe()` then creates a `uprobe_kimg` for the probepoint, queues it for insertion, requests (via utrace) that the probed process "quiesce," sleeps until the insertion takes place, and then returns. (If there's already a probepoint at the specified address, `register_uprobe()` just adds the uprobe to that `uprobe_kimg` and returns.)

Note that since all threads in a process share the same text, there's no way to register a uprobe for a particular thread in a multithreaded process.

Once all threads in the probed process have quiesced, the last thread to quiesce inserts a breakpoint instruction at the specified probepoint, rouses the quiesced threads, and wakes up `register_uprobe()`.

`unregister_uprobe()` works in reverse: Queue the `uprobe_kimg` for removal, quiesce the probed process, sleep until the probepoint has been removed, and delete the `uprobe_kimg`. If this was the last `uprobe_kimg` for the process, `unregister_uprobe()` tears down the entire `uprobe_process`, along with its `uprobe_tasks`.

5.3 Utrace callbacks

Aside from registration and unregistration, everything in uprobes happens as the result of a utrace callback. When a `uprobe_task` is created, uprobes calls `utrace_attach()` to create an engine for that thread, and listens for the following events in that task:

- *breakpoint signal* (SIGTRAP on most architectures): Utrace notifies uprobes when the probed task hits a breakpoint. Uprobes determines which probepoint (if any) was hit, runs the associated probe handler(s), and directs utrace to single-step the probed instruction.

- *single-step signal* (SIGTRAP on most architectures): Utrace notifies uprobes after the probed instruction has been single-stepped. Uprobes does any necessary post-single-step work (discussed in later sections), and directs utrace to continue execution of the probed task.
- *fork/clone:* Utrace notifies uprobes when a probed task forks a new process or clones a new thread. When a new thread is cloned for the probed process, uprobes adds a new `uprobe_task` to the `uprobe_process`, complete with an appropriately programmed utrace engine. In the case of `fork()`, uprobes doesn't attempt to preserve probepoints in the child process, since each uprobe object refers to only one process. Rather, uprobes iterates through all the probe addresses in the parent and removes the breakpoint instructions in the child.
- *exit:* Utrace notifies uprobes when a probed task exits. Uprobes deletes the corresponding `uprobe_task`. If this was the last `uprobe_task` for the process, uprobes tears down the entire `uprobe_process`. (Since the process is exiting, there's no need to remove breakpoints.)
- *exec:* Utrace notifies uprobes when a probed task execs a new program. (Utrace reports exit events for any other threads in the process.) Since probepoints in the old program are irrelevant in the new program, uprobes tears down the entire `uprobe_process`. (Again, there's no need to remove breakpoints.)
- *quiesce:* Uprobes listens for the above-listed events all the time. By contrast, uprobes listens for quiesce events only while it's waiting for the probed process to quiesce, in preparation for insertion or removal of a breakpoint instruction. (See Section 6.2).

5.4 Architecture-specific code

Most components of the architecture-specific code for uprobes are simple macros and inline functions. Support for return probes (uretprobes) typically adds 10–40 lines. By far the majority of the architecture-specific code relates to "fix-ups" necessitated by the fact that we single-step a copy of the probed instruction, rather than single-stepping it in place. See "Tracing multithreaded applications: SSOL" in the next section.

6 Uprobes Implementation Notes

6.1 Tracing multithreaded applications: SSOL

Like some other tracing and debugging tools, uprobes implements a probepoint by replacing the first byte(s) of the instruction at the probepoint with a breakpoint instruction, after first saving a copy of the original instruction. After the breakpoint is hit and the handler has been run, uprobes needs to execute the original instruction in the context of the probed process. There are two commonly accepted ways to do this:

- *Single-stepping inline (SSIL):* Temporarily replace the breakpoint instruction with the original instruction; single-step the instruction; restore the breakpoint instruction; and allow the task to continue. This method is typically used by interactive debuggers such as gdb.
- *Single-stepping out of line (SSOL):* Place a copy of the original instruction somewhere in the probed process's address space; single-step the copy; "fix up" the task's state as necessary; and allow the task to continue. If the effect of the instruction depends on its address (e.g., a relative branch), the task's registers and/or stack must be "fixed up" after the instruction is executed (e.g., to make the program counter relative to the address of the original instruction, rather than the instruction copy). This method is used by kprobes for kernel tracing.

The SSIL approach doesn't work acceptably for multithreaded programs. In particular, while the breakpoint instruction is temporarily removed during single-stepping, another thread can sail past the probepoint. We considered the approach of quiescing, or otherwise blocking, all threads every time one hits a probepoint, so that we could be sure of no probe misses during SSIL, but we considered the performance implications unacceptable.

In terms of implementation, SSOL has two important implications:

- Each uprobes port must provide code to do architecture-specific post-single-step fix-ups. Much of this code can be filched from kprobes,

but there are additional implications for uprobes. For example, uprobes must be prepared to handle any instruction in the architecture's instruction set, not just those used in the kernel; and for some architectures, uprobes must be able to handle both 32-bit and 64-bit user applications.

- The instruction copy to be single-stepped must reside somewhere in the probed process's address space. Since uprobes can't know what other threads may be doing while a thread is single-stepping, the instruction copy can't reside in any location legitimately used by the program. After considering various approaches, we decided to allocate a new VM area in the probed process to hold the instruction copies. We call this the *SSOL area*.

To minimize the impact on the system, uprobes allocates the SSOL area only for processes that are actually probed, and the area is small (typically one page) and of fixed size. "Instruction slots" in this area are allocated on a per-probepoint basis, so that multiple threads can single-step in the same slot simultaneously. Slots are allocated only to probepoints that are actually hit. If uprobes runs out of free slots, slots are recycled on a least-recently-used basis.

6.2 Quiescing

Uprobes takes a fairly conservative approach when inserting and removing breakpoints: all threads in the probed process must be "quiescent" before the breakpoint is inserted/removed.

Our approach to quiescing the threads started out fairly simple: For each task in the probed process, the `[un]register_uprobe()` function sets the `UTRACE_ACTION_QUIESCE` flag in the `uprobe_task`'s engine, with the result that utrace soon brings the task to a stopped state and calls uprobes's quiesce callback for that task. After setting all tasks on the road to quiescence, `[un]register_uprobe()` goes to sleep. For the last thread to quiesce, the quiesce callback inserts or removes the requested breakpoint, wakes up `[un]register_uprobe()`, and rouses all the quiesced threads.

It turned out to be more complicated than that. For example:

- If the targeted thread is already quiesced, setting the `UTRACE_ACTION_QUIESCE` flag causes the quiesce callback to run immediately in the context of the `[un]register_uprobe()` task. This breaks the "sleep until the last quiescent thread wakes me" paradigm.

- If a thread hits a breakpoint on the road to quiescence, its quiesce callback is called before the signal callback. This turns out to be a bad place to stop for breakpoint insertion or removal. For example, if we happen to remove the `uprobe_kimg` for the probepoint we just hit, the subsequent signal callback won't know what to do with the SIGTRAP.

- If `[un]register_uprobe()` is called from a uprobe handler, it runs in the context of the probed task. Again, this breaks the "sleep until the last quiescent thread wakes me" paradigm.

It turned out to be expedient to establish an "alternate quiesce point" in uprobes, in addition to the quiesce callback. When it finishes handling a probepoint, the uprobes signal callback checks to see whether the process is supposed to be quiescing. If so, it does essentially what the quiesce callback does: if it's the last thread to "quiesce," it processes the pending probe insertion or removal and rouses the other threads; otherwise, it sleeps in a pseudo-quiesced state until the "last" thread rouses it. Consequently, if `[un]register_uprobe()` sees that a thread is currently processing a probepoint, it doesn't try to quiesce it, knowing that it will soon hit the alternate quiesce point.

6.3 User-space return probes

Uprobes supports a second type of probe: a return probe fires when a specified function returns. User-space return probes (uretprobes) are modeled after return probes in kprobes (kretprobes).

When you register a uretprobe, you specify the process, the function (i.e., the address of the first instruction), and a handler to be run when the function returns.

Uprobes sets a probepoint at the entry to the function. When the function is called, uprobes saves a copy of the return address (which may be on the stack on in a register, depending on the architecture) and replaces the return address with the address of the "uretprobe trampoline," which is simply a breakpoint instruction.

When the function returns, control passes to the trampoline, the breakpoint is hit, and uprobes gains control. Uprobes runs the user-specified handler, then restores the original return address and allows the probed function to return.

In uprobes, the return-probes implementation differs from kprobes in several ways:

- The user doesn't need to specify how many "return-probe instance" objects to preallocate. Since uprobes runs in a context where it can use `kmalloc()` freely, no preallocation is necessary.
- Each probed process needs a trampoline in its address space. We use one of the slots in the SSOL area for this purpose.
- As in kprobes, it's permissible to unregister the return probe while the probed function is running. Even after all probes have been removed, uprobes keeps the `uprobe_process` and its `uprobe_tasks` (and utrace engines) around as long as necessary to catch and process the last hit on the uretprobe trampoline.

6.4 Registering/unregistering probes in probe handlers

A uprobe or uretprobe handler can call any of the functions in the uprobes API. A handler can even unregister its own probe. However, when invoked from a handler, the actual [un]register operations do not take place immediately. Rather, they are queued up and executed after all handlers for that probepoint have been run and the probed instruction has been single-stepped. (Specifically, queued [un]registrations are run right after the previously described "alternate quiesce point.") If the `registration_callback` field is set in the uprobe object to be acted on, uprobes calls that callback when the [un]register operation completes.

An instrumentation module that employs such dynamic [un]registrations needs to keep track of them: since a module's uprobe objects typically disappear along with the module, the module's cleanup function should not exit while any such operations are outstanding.

7 Applying Uprobes

We envision uprobes being used in the following situations, for debugging and/or for performance monitoring:

- Tracing timing-sensitive applications.
- Tracing multithreaded applications.
- Tracing very large and/or complex applications.
- Diagnosis of systemic performance problems involving multiple layers of software (in kernel and user space).
- Tracing applications in creative ways – e.g., collecting different types of information at different probepoints, or dynamically adjusting which code points are probed.

8 Future Work

What's next for utrace, uprobes, and kprobes? Here are some possibilities:

- *Utrace += SSOL.* As discussed in Section 6.1, single-stepping out of line is crucial for the support of probepoints in multithreaded processes. This technology may be migrated from uprobes to utrace, so that other utrace clients can exploit it. One such client might be an enhanced ptrace.
- *SystemTap += utrace + uprobes.* Some SystemTap users want support for probing in user space. Some potential utrace and uprobes users might be more enthusiastic given the safety and ease of use provided by SystemTap.
- *Registering probes from kprobe handlers.* Utrace and uprobe handlers can register and unregister utrace, uprobes, and kprobes handlers. It would be nice if kprobes handlers could do the same. Perhaps the effect of sleep-tolerant kprobe handlers could be approximated using a kernel thread that runs deferred handlers. This possibility is under investigation.
- *Tracing Java.* Uprobes takes us closer to dynamic tracing of the Java Virtual Machine and Java applications.

- *Task-independent exec hook.* Currently, uprobes can trace an application if it's already running, or if it is known which process will spawn it. Allowing tracing of applications that are yet be started and are of unknown lineage will help to solve problems that creep in during application startup.

9 Acknowledgements

The authors wish to thank Roland McGrath for coming up with the excellent utrace infrastructure, and also for providing valuable suggestions and feedback on uprobes. Thanks are due to David Wilder and Srikar Dronamraju for helping out with testing the uprobes framework. Thanks also to Dave Hansen and Andrew Morton for their valuable suggestions on the VM-related portions of the uprobes infrastructure.

10 Legal Statements

References

[1] Suparna Bhattacharya. Dynamic Probes - Debugging by Stealth. In *Proceedings of Linux.Conf.Au*, 2003.

[2] Mavinakayanahalli et al. Probing the Guts of Kprobes. In *Proceedings of the Ottawa Linux Symposium*, 2006.

[3] Roland McGrath. Utrace home page. `http://people.redhat.com/roland/utrace/`.

[4] SystemTap project team. SystemTap. `http://sourceware.org/systemtap/`.

kvm: the Linux Virtual Machine Monitor

Avi Kivity
Qumranet
avi@qumranet.com

Yaniv Kamay
Qumranet
yaniv@qumranet.com

Dor Laor
Qumranet
dor.laor@qumranet.com

Uri Lublin
Qumranet
uril@qumranet.com

Anthony Liguori
IBM
aliguori@us.ibm.com

Abstract

Virtualization is a hot topic in operating systems these days. It is useful in many scenarios: server consolidation, virtual test environments, and for Linux enthusiasts who still can not decide which distribution is best. Recently, hardware vendors of commodity x86 processors have added virtualization extensions to the instruction set that can be utilized to write relatively simple virtual machine monitors.

The Kernel-based Virtual Machine, or **kvm**, is a new Linux subsystem which leverages these virtualization extensions to add a virtual machine monitor (or hypervisor) capability to Linux. Using **kvm**, one can create and run multiple virtual machines. These virtual machines appear as normal Linux processes and integrate seamlessly with the rest of the system.

1 Background

Virtualization has been around almost as long as computers. The idea of using a computer system to emulate another, similar, computer system was early recognized as useful for testing and resource utilization purposes. As with many computer technologies, IBM led the way with their VM system. In the last decade, VMware's software-only virtual machine monitor has been quite successful. More recently, the Xen [xen] open-source hypervisor brought virtualization to the open source world, first with a variant termed *paravirtualization* and as hardware became available, full virtualization.

2 x86 Hardware Virtualization Extensions

x86 hardware is notoriously difficult to virtualize. Some instructions that expose privileged state do not trap when executed in user mode, e.g. *popf*. Some privileged state is difficult to hide, e.g. the current privilege level, or *cpl*.

Recognizing the importance of virtualization, hardware vendors [Intel][AMD] have added extensions to the x86 architecture that make virtualization much easier. While these extensions are incompatible with each other, they are essentially similar, consisting of:

- A new guest operating mode – the processor can switch into a guest mode, which has all the regular privilege levels of the normal operating modes, except that system software can selectively request that certain instructions, or certain register accesses, be trapped.
- Hardware state switch – when switching to guest mode and back, the hardware switches the control registers that affect processor operation modes, as well as the segment registers that are difficult to switch, and the instruction pointer so that a control transfer can take effect.
- Exit reason reporting – when a switch from guest mode back to host mode occurs, the hardware reports the reason for the switch so that software can take the appropriate action.

3 General `kvm` Architecture

Under `kvm`, virtual machines are created by opening a device node (`/dev/kvm`.) A guest has its own memory, separate from the userspace process that created it. A virtual cpu is not scheduled on its own, however.

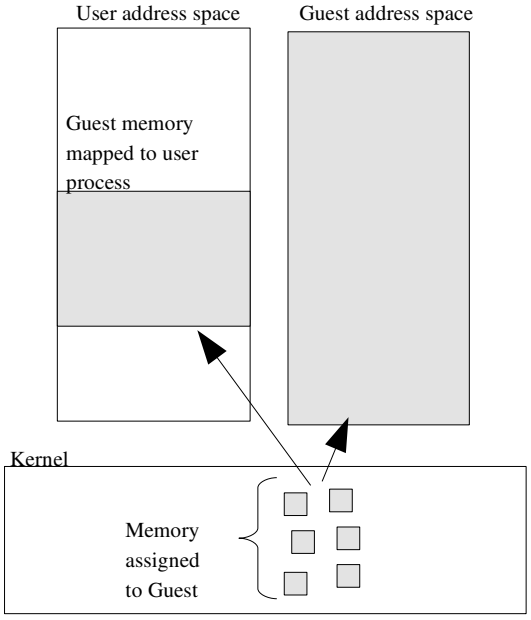


Figure 1: `kvm` Memory Map

3.1 `/dev/kvm` Device Node

`kvm` is structured as a fairly typical Linux character device. It exposes a `/dev/kvm` device node which can be used by userspace to create and run virtual machines through a set of `ioctl()`s.

The operations provided by `/dev/kvm` include:

- Creation of a new virtual machine.
- Allocation of memory to a virtual machine.
- Reading and writing virtual cpu registers.
- Injecting an interrupt into a virtual cpu.
- Running a virtual cpu.

Figure 1 shows how guest memory is arranged. Like user memory in Linux, the kernel allocates discontiguous pages to form the guest address space. In addition, userspace can `mmap()` guest memory to obtain direct access. This is useful for emulating dma-capable devices.

Running a virtual cpu deserves some further elaboration. In effect, a new execution mode, *guest mode* is added to Linux, joining the existing *kernel mode* and *user mode*.

Guest execution is performed in a triply-nested loop:

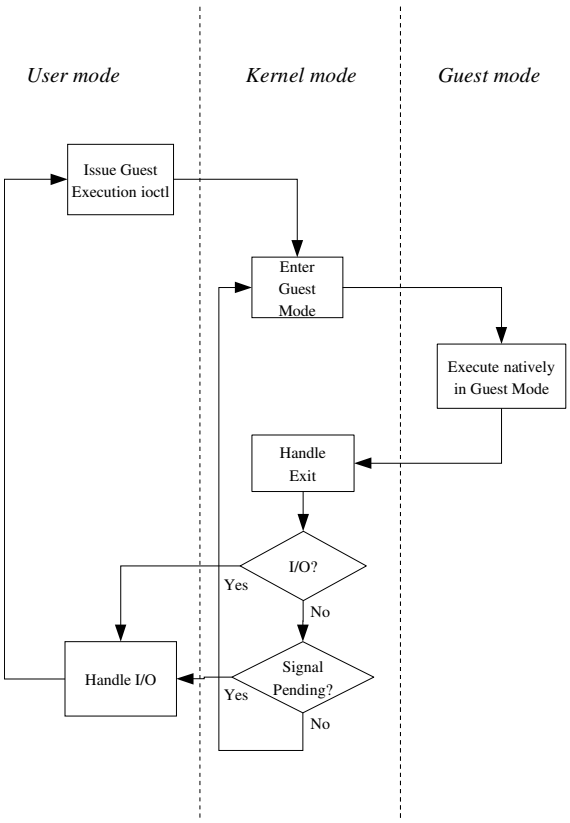


Figure 2: Guest Execution Loop

- At the outermost level, userspace calls the kernel to execute guest code until it encounters an I/O instruction, or until an external event such as arrival of a network packet or a timeout occurs. External events are represented by signals.

- At the kernel level, the kernel causes the hardware to enter guest mode. If the processor exits guest mode due to an event such as an external interrupt or a shadow page table fault, the kernel performs the necessary handling and resumes guest execution. If the exit reason is due to an I/O instruction or a signal queued to the process, then the kernel exits to userspace.

- At the hardware level, the processor executes guest code until it encounters an instruction that needs assistance, a fault, or an external interrupt.

Refer to Figure 2 for a flowchart-like representation of the guest execution loop.

3.2 Reconciling Instruction Set Architecture Differences

Unlike most of the x86 instruction set, where instruction set extensions introduced by one vendor are adopted by the others, hardware virtualization extensions are not standardized. Intel and AMD processors have different instructions, different semantics, and different capabilities.

`kvm` handles this difference in the traditional Linux way of introducing a function pointer vector, `kvm_arch_ops`, and calling one of the functions it defines whenever an architecture-dependent operation is to be performed. Base `kvm` functionality is placed in a module, `kvm.ko`, while the architecture-specific functionality is placed in the two arch-specific modules, `kvm-intel.ko` and `kvm-amd.ko`.

4 Virtualizing the MMU

As with all modern processors, x86 provides a virtual memory system which translates user-visible virtual addresses to physical addresses that are used to access the bus. This translation is performed by the *memory management unit*, or *mmu*. The mmu consists of:

- A radix tree ,the *page table*, encoding the virtual-to-physical translation. This tree is provided by system software on physical memory, but is rooted in a hardware register (the `cr3` register)
- A mechanism to notify system software of missing translations (page faults)
- An on-chip cache (the *translation lookaside buffer*, or *tlb*) that accelerates lookups of the page table
- Instructions for switching the translation root in order to provide independent address spaces
- Instructions for managing the tlb

The hardware support for mmu virtualization provides hooks to all of these components, but does not fully virtualize them. The principal problem is that the mmu provides for one level of translation ($guest virtual \rightarrow guest physical$) but does not account for the second level required by virtualization ($guest physical \rightarrow host physical$.)

The classical solution is to use the hardware virtualization capabilities to present the real mmu with a separate page table that encodes the combined translation ($guest virtual \rightarrow host physical$) while emulating the hardware's interaction with the original page table provided by the guest. The shadow page table is built incrementally; it starts out empty, and as translation failures are reported to the host, missing entries are added.

A major problem with shadow page tables is keeping the guest page table and the shadow page table synchronized. Whenever the guest writes to a page table, the corresponding change must also be performed on the shadow page table. This is difficult as the guest page table resides in ordinary memory and thus is not normally trapped on access.

4.1 Virtual TLB Implementation

The initial version of shadow page tables algorithm in `kvm` used a straightforward approach that reduces the amount of bugs in the code while sacrificing performance. It relies on the fact that the guest must use the tlb management instructions to synchronize the tlb with its page tables. We trap these instructions and apply their effect to the shadow page table *in addition* to the normal effect on the tlb.

Unfortunately, the most common tlb management instruction is the context switch, which effectively invalidates the entire tlb.[1] This means that workloads with multiple processes suffer greatly, as rebuilding the shadow page table is much more expensive than refilling the tlb.

4.2 Caching Virtual MMU

In order to improve guest performance, the virtual mmu implementation was enhanced to allow page tables to be cached across context switches. This greatly increases performance at the expense of much increased code complexity.

As related earlier, the problem is that guest writes to the guest page tables are not ordinarily trapped by the virtualization hardware. In order to receive notifications of such guest writes, we *write-protect* guest memory pages that are shadowed by `kvm`. Unfortunately, this causes a chain reaction of additional requirements:

[1] Actually, kernel mappings can be spared from this flush; but the performance impact is nevertheless great.

- To write protect a guest page, we need to know which translations the guest can use to write to the page. This means we need to keep a *reverse mapping* of all writable translations that point to each guest page.
- When a write to a guest page table is trapped, we need to emulate the access using an x86 instruction interpreter so that we know precisely the effect on both guest memory and the shadow page table.
- The guest may recycle a page table page into a normal page without a way for `kvm` to know. This can cause a significant slowdown as writes to that page will be emulated instead of proceeding at native speeds. `kvm` has heuristics that determine when such an event has occurred and decache the corresponding shadow page table, eliminating the need to write-protect the page.

At the expense of considerable complexity, these requirements have been implemented and `kvm` context switch performance is now reasonable.

5 I/O Virtualization

Software uses *programmed I/O* (*pio*) and *memory-mapped I/O* (*mmio*) to communicate with hardware devices. In addition, hardware can issue *interrupts* to request service by system software. A virtual machine monitor must be able to trap and emulate pio and mmio requests, and to simulate interrupts from virtual hardware.

5.1 Virtualizing Guest-Initiated I/O Instructions

Trapping pio is quite straightforward as the hardware provides traps for pio instructions and partially decodes the operands. Trapping mmio, on the other hand, is quite complex, as the same instructions are used for regular memory accesses and mmio:

- The `kvm` mmu does *not* create a shadow page table translation when an mmio page is accessed
- Instead, the x86 emulator executes the faulting instruction, yielding the direction, size, address, and value of the transfer.

In `kvm`, I/O virtualization is performed by userspace. All pio and mmio accesses are forwarded to userspace, which feeds them into a *device model* in order to simulate their behavior, and possibly trigger real I/O such as transmitting a packet on an Ethernet interface. `kvm` also provides a mechanism for userspace to inject interrupts into the guest.

5.2 Host-Initiated Virtual Interrupts

`kvm` also provides interrupt injection facilities to userspace. Means exist to determine when the guest is ready to accept an interrupt, for example, the interrupt flag must be set, and to actually inject the interrupt when the guest is ready. This allows `kvm` to emulate the APIC/PIC/IOAPIC complex found on x86-based systems.

5.3 Virtualizing Framebuffers

An important category of memory-mapped I/O devices are framebuffers, or graphics adapters. These have characteristics that are quite distinct from other typical mmio devices:

- *Bandwidth* – framebuffers typically see very high bandwidth transfers. This is in contrast to typical devices which use mmio for control, but transfer the bulk of the data with direct memory access (*dma*).
- *Memory equivalence* – framebuffers are mostly just memory: reading from a framebuffers returns the data last written, and writing data does not cause an action to take place.

In order to efficiently support framebuffers, `kvm` allows mapping non-mmio memory at arbitrary addresses such as the pci region. Support is included for the VGA windows which allow physically aliasing memory regions, and for reporting changes in the content of the framebuffer so that the display window can be updated incrementally.

6 Linux Integration

Being tightly integrated into Linux confers some important benefits to `kvm`:

- On the *developer* level, there are many opportunities for reusing existing functionality within the kernel, for example, the scheduler, NUMA support, and high-resolution timers.
- On the *user* level, one can reuse the existing Linux process management infrastructure, e.g., `top(1)` to look at cpu usage and `taskset(1)` to pin virtual machines to specific cpus. Users can use `kill(1)` to pause or terminate their virtual machines.

7 Live Migration

One of the most compelling reasons to use virtualization is *live migration*, or the ability to transport a virtual machine from one host to another without interrupting guest execution for more than a few tens of milliseconds. This facility allows virtual machines to be relocated to different hosts to suit varying load and performance requirements.

Live migration works by copying guest memory to the target host in parallel with normal guest execution. If a guest page has been modified *after* it has been copied, it must be copied again. To that end, `kvm` provides a *dirty page log* facility, which provides userspace with a bitmap of modified pages since the last call. Internally, `kvm` maps guest pages as read-only, and only maps them for write after the first write access, which provides a hook point to update the bitmap.

Live migration is an iterative process: as each pass copies memory to the remote host, the guest generates more memory to copy. In order to ensure that the process converges, we set the following termination criteria:

- Two, not necessarily consecutive, passes were made which had an *increase* in the amount of memory copied compared to previous pass, or,
- Thirty iterations have elapsed.

8 Future Directions

While already quite usable for many workloads, many things remain to be done for `kvm`. Here we describe the major features missing; some of them are already work-in-progress.

8.1 Guest SMP Support

Demanding workloads require multiple processing cores, and virtualization workloads are no exception. While `kvm` readily supports SMP hosts, it does not yet support SMP guests.

In the same way that a virtual machine maps to a host process under `kvm`, a virtual cpu in an SMP guest maps to a host thread. This keeps the simplicity of the `kvm` model and requires remarkably few changes to implement.

8.2 Paravirtualization

I/O is notoriously slow in virtualization solutions. This is because emulating an I/O access requires exiting guest mode, which is a fairly expensive operation compared to real hardware.

A common solution is to introduce paravirtualized devices, or virtual "hardware" that is explicitly designed for virtualized environments. Since it is designed with the performance characteristics of virtualization in mind, it can minimize the slow operations to improve performance.

8.3 Memory Management Integration

Linux provides a vast array of memory management features: demand paging, large pages (*hugetlbfs*), and memory-mapped files. We plan to allow a `kvm` guest address space to directly use these features; this can enable paging of idle guest memory to disk, or loading a guest memory image from disk by demand paging.

8.4 Scheduler Integration

Currently, the Linux scheduler has no knowledge that it is scheduling a virtual cpu instead of a regular thread. We plan to add this knowledge to the scheduler so that it can take into account the higher costs of moving a virtual cpu from one core to another, as compared to a regular task.

8.5 New Hardware Virtualization Features

Virtualization hardware is constantly being enhanced with new capabilities, for example, full mmu virtualization, a.k.a. *nested page tables* or *extended page tables*, or allowing a guest to securely access a physical device [VT-d]. We plan to integrate these features into `kvm` in order to gain the performance and functionality benefits.

8.6 Additional Architectures

`kvm` is currently only implemented for the `i386` and `x86-64` architectures. However, other architectures such as `powerpc` and `ia64` support virtualization, and `kvm` could be enhanced to support these architectures as well.

9 Conclusions

`kvm` brings an easy-to-use, fully featured integrated virtualization solution for Linux. Its simplicity makes extending it fairly easy, while its integration into Linux allows it to leverage the large Linux feature set and the tremendous pace at which Linux is evolving.

10 References

[qemu] Bellard, F. (2005). *Qemu, a Fast and Portable Dynamic Translator.* In Usenix annual technical conference.

[xen] Barham P., et al. *Xen and the art of virtualization.* In Proc. SOSP 2003. Bolton Landing, New York, U.S.A. Oct 19-22, 2003.

[Intel] Intel Corp. *IA-32 Intel® Architecture Software Developer's Manual, Volume 3B: System Programming Guide, Part 2.* Order number 25366919.

[AMD] AMD Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming.*

[VT-d] Abramson, D.; Jackson, J.; Muthrasanallur, S.; Neiger, G.; Regnier, G.; Sankaran, R.; Schoinas, I.; Uhlig, R.; Vembu, B.; Wiegert, J. *Intel® Virtualization Technology for Directed I/O.* Intel Technology Journal. `http://www.intel.com/technology/itj/2006/v10i3/` (August 2006).

Linux Telephony

A Short Overview

Paul P. Komkoff
Google Ireland Ltd.
i@stingr.net

Anna Anikina
Saratov State University
anna@sgu.ru

Roman Zhnichkov
Saratov State University
zr@sgu.ru

Abstract

This paper covers the topic of implementing voice services in packet-switched Voice Over IP (VoIP) and circuit-switched (traditional telephony) networks using Linux and commodity hardware. It contains a short introduction into general telephony, VoIP, and Public Switched Telephony Networks (PSTN). It also provides an overview of VoIP services, including the open-source software packages used to implement them, and the kernel interfaces they include. It explains kernel support for connecting Public Switched Telephony Networks using digital interfaces (E1/T1) via the *Zaptel* framework, and user-space integration issues. The conclusion examines current trends in Linux-based and open-source telephony.

A basic understanding of networking concepts is helpful for understanding this presentation.

1 Introduction to general telephony, VoIP and PSTN

Although VoIP products like *Skype*™ and *Google Talk*™ are storming the telecommunications landscape, almost everyone still owns a telephone and uses it daily. Understanding how telephony works is still an obscure topic for most people. As a result, we would like to begin our paper with a short introduction to telephony in general, combined with two major types presently in use—packet-switched and circuit-switched.

Every telephony system needs to transmit data between parties, but before it can do so, it needs to find the other party and a route to it. This is called *call setup*. Activity of this nature is usually performed by a *signalling protocol*. Data can be passed via the same route and channel as signalling, or via the same route and a different channel, or via an entirely different route. There are obvious *quality of service* (QOS) requirements for signalling and data—signalling requires guaranteed delivery of every message, and data requires low-latency transmission, but can lose individual samples/frames.

1.1 VoIP

Voice over IP (VoIP) is widely used to describe various services, setups, and protocols that pass audio data in pseudo real-time over IP networks. Although the actual implementations are very different, the basic requirements are to pass voice data in two directions and to allow two-party conversation. VoIP is *packet-switched telephony* because the underlying network is packet-switched. To make conversations possible a virtual circuit is built between parties.

There are many protocols used for VoIP conversations. The most widespread is Session Initiation Protocol (SIP). This is a signalling protocol in that it only handles call setup and termination. Actual session setup between endpoints is handled by *SDP* (Session Description Protocol), and data is transmitted by *RTP* (Real-Time Protocol). SIP is described in RFC3561 and endorsed by IETF as an Internet standard.

Another protocol with a large user-base is *H.323*. While SIP is designed mostly by network people it is similar to HTTP (its messages are text `Name:Value` pairs). H.323, being endorsed by *ITU*—telephony people—looks like a traditional telephony protocol. Its messages and information elements are described in *ASN.1* (Abstract Syntax Notation) and coded by ASN.1 *BER* (Basic Encoding Rules). H.323 is also a signalling protocol, and also uses RTP for data transmission.

Inter-Asterisk Exchange (IAX) is another major VoIP protocol. It was invented by the *Asterisk*® authors. In this protocol, data and signalling streams are not separated, which allows easier NAT traversal. IAX is also

able to pack multiple conversations into a single stream, thus reducing trunking overhead. There are two versions of IAX, although IAX1 is deprecated, and IAX2 is used under name of IAX.

Other lesser known VoIP protocols are either proprietary (*SCCP* is used in Cisco phones) or designed for specific purpose (*MGCP*—Media Gateway Control Protocol—used to manipulate VoIP gateways). Some protocols allow direct conversation between *endpoints* while others require a server to operate.

VoIP protocols define the way voice data should be transmitted. Conversion of digitized audio into *payload* (portions of data to be delivered by the VoIP protocol) is performed by *voice codecs*. The most popular voice codecs are: G.711, G.723, G.726-G.729, iLBC, GSM06.10, and Speex. Each has different bandwidth requirements, CPU requirements, quality, and patents associated with them. These codecs are not all VoIP-specific—G.711 is used in traditional telephony, and GSM06.10 is used in GSM mobile networks.

1.2 PSTN

Public Switched Telephony Network (PSTN) is the traditional telephony network. Capable nodes in this network are addressed using global and unique *E.164 addresses*—telephone numbers. Nowadays PSTN includes not only traditional telephones, but mobile phones as well. Most of the PSTN is digital (except customer analog lines and very old installments in developing countries). PSTN is *circuit-switched*, which means that the call setup procedure assembles a circuit between the parties on the call for the duration of the entire call. The circuit is either fixed-bandwidth digital (typically 64kbps for traditional telephone networks and 9600bps for mobile networks) or analog—spanning multiple units of communication equipment. In digital networks the circuit is made over *E1* (for Europe) or *T1* (America) lines, which contain 31 or 24 DS0 (64kbps) circuits, TDM-multiplexed together and thus often called *timeslots*.

The call management, call routing, circuit assignment and maintenance procedures are performed by the *signaling protocol*. The de facto standard for interconnecting networks is Signaling System 7 (SS7). Connections to customer *PBX* (Private Branch Exchange) are often performed using ISDN PRI (Primary Rate Interface) connections and ISDN Q.931 signaling. SS7 is not a single protocol, but a protocol stack. It contains *parts* which facilitate running SS7 itself and allows *user parts* to run on top of it. The user part that is responsible for voice call setup between parties is called *ISUP* (ISDN User Part). Services for mobile subscribers, ranging from registration to SMS, are handled by *MAP* over *TCAP* (Mobile Application Part over Transaction Capabilities Application Part) of SS7.

2 Implementing VoIP services on Linux

To implement any voice service using VoIP, we do not need any special hardware. Both clients and servers are done in software. We only need to implement a particular VoIP protocol (SIP, H.323, IAX, MGCP, SCCP) and a particular set of voice codecs.

After we have everything ready on the protocol and codec sides, we can implement the voice service. For example, we can build a server that will handle client registrations and calls to each other. This piece of software is typically called a *softswitch* because it functions much like a hardware switch—building virtual circuits between parties. Softswitches typically have the ability to provide optional services commonly found in traditional proprietary PBXes like conferencing, voicemail, and *IVR* (Interactive Voice Responce) for an additional charge. Modern opensource VoIP software logic is driven by powerful scripting languages—domain-specific (for building dialplans) or general purpose. This allows us to integrate with almost anything. For example, we can try opensource speech synthesis/recognition software.

Many softswitches utilize special properties of particular VoIP protocols. For example, SIP and the H.323 architecture provide the ability to pass data directly between endpoints to reduce contention and minimize latency. Thousands of endpoints can be registered to one SIP server to control only signalling, allowing billing and additional services. This is much better than sitting in the middle of RTP streams between those clients. Moreover, sometimes it is possible to pass data directly between two endpoints while one of them is using SIP and another—H.323. This setup is called a *signalling proxy*.

Some softswitches are suitable only for VoIP clients (including VoIP-PSTN gateways acting as VoIP endpoint) while more general solutions are able to act as a switch

between different VoIP protocols and PSTN itself. The softswitches of the first kind are, for example, `SIP Express Router` and `sipX`, while the major players of the second kind are: `Asterisk`, `CallWeaver` (described later on), `YATE`, and `FreeSwitch`.

To test the most widespread softswitch—`Asterisk`—using VoIP only, download its source code from `http://asterisk.org`, compile it, and install. Without any additional software you have out-of-the-box support for SIP and IAX, a working IVR demo (in `extensions.conf` file), and many functions which you can attach to numbers in extensions.conf—*asterisk applications*. However, conferencing won't work and music-on-hold can stutter.

Call-center solutions based on Asterisk usually utilize `Queue()` application. Using different *AGI* (Asterisk Gateway Interface) scripts and builtin applications like `SayNumber()`, you can build an automatic answering machine which reports the current time or account balance. Asterisk can make outgoing calls as well if a specifically formatted text-file is added to the special directory for each call.

Another software package to try is Yate. Its architecture is different, however, you still can easily test basic functions of an IP PBX. Yate can be configured to be a signalling proxy between H.323 and SIP—a desired usage when building VoIP exchanges.

What does *Linux support for VoIP* mean here? It means fast, capable UDP (and ioctls which permit setting a particular DSCP on outgoing packets), a CPU scheduler which will not starve us receiving (if we are using blocking/threaded model), sending, and processing, and a preemptive kernel to reduce receive latency. However, there are still problems when a large number of clients are passing data through a single server.

Recent improvements in the kernel, namely in the scheduling, preemption, and high-precision timers have greatly improved its ability to run userspace telephony applications.

2.1 VoIP clients

There are two primary types of VoIP clients or *endpoints*—those running on dedicated hardware (handsets plugged into Ethernet, analog telephone adapters, dedicated VoIP gateways), and softphones—installable applications for your favorite operating system.

Popular open source softphones include: `Ekiga` (previously Gnome Meeting), `Kphone`, and `Kiax`, which support major protocols (H.323, SIP, and IAX). The supported voice codecs list is not as long as it might be due to patent issues. Even with access to an entirely free alternative like Speex, the user is forced to use patented codecs to connect to proprietary VoIP gateways and consumer devices.

Skype™, a very popular proprietary softphone, implements its own proprietary VoIP protocol.

3 Connecting to PSTN

In order to allow PSTN users to use the services described above, or at a minimum send and receive calls from other VoIP users, they need to connect to the PSTN. There are several ways to do that:

- analog connection, either FXO (office) or FXS (station) side
- ISDN BRI (Basic Rate Interface) connection
- ISDN PRI or SS7 on E1/T1 line

We will concentrate on the most capable way to connect a E1/T1 digital interface (supporting ISDN PRI or SS7 directly) to a VoIP server. Carrier equipment is interconnected in this way. E1/T1-capable hardware and kernel is required to support this.

The "original" digital telephony interface cards compatible with Asterisk are manufactured by Digium®. Each contains up to 4 E1/T1/J1 ports. Other manufacturers have also unveiled similar cards, namely Sangoma and Cronyx. Clones of the Digium cards are also available in the wild (OpenVox) which behave in exactly the same way as the Digium ones.

One way to present telephony interfaces to an application is by using the *Zaptel* framework. The official zaptel package, released by Digium together with Asterisk, contains the zaptel framework and drivers for Digium and Digium-endorsed hardware. Although drivers for other mentioned hardware have different architectures, they implement zaptel hooks and are thus compatible (to a certain extent) with the largest user base of such equipment. However, other software can use other ways

of interacting with hardware. For example, Yate (partially sponsored by Sangoma), can use native ways of communicating with Sangoma cards. Extremely high performance has been shown in those setups.

After you have ISDN PRI provisioned to you and a Digium card at hand, obtain the latest Zaptel drivers (also at `http://asterisk.org`) and compile them. If everything goes well and you successfully insmod (load) the right modules (and you have udev), you will notice a couple of new device nodes under `/dev/zap`. Before starting any telephony application, you need to configure zaptel ports using `ztcfg` tool. After configuration you will have additional nodes `/dev/zap/X`, one for each channel you configured. In ISDN PRI, the 16th timeslot of E1 is dedicated signalling channel (D-chan). As a result it runs Q.931 over Q.921 over HDLC. All other timeslots are clear-channels (B-chan) and are used to transmit data. At a minimum, the ISDN PRI application needs to talk Q.931 over the D-channel, negotiate B-channel number for conversations, and read/write digitized audio data from/to the specific B-channel.

Achieving SS7 connectivity is slightly more difficult. Until 2006, there was no working opensource SS7 implementation. Even today, you still need to find a carrier who will allow an uncertified SS7 device on their network. On the other hand, when you are *the* carrier, having opensource SS7 is extremely useful for a number of reasons. One might be your traditional PSTN switch—which has only SS7 ports free when buying ISDN PRI ports isn't an option.

Today there is at least one usable SS7 implementation for asterisk—Sifira's `chan_ss7`, available at `http://www.sifira.dk/chan-ss7/`. An opensource SS7 stack for Yate (yss7) is in progress.

What kind of services can we implement here? VoIP-PSTN gateway? Indeed, if we are able to capture the voice onto the system, we can transmit and receive it over the network. Because we use an opensource softswitch for this purpose, we get a full-fledged IP-PBX with PSTN interface, capable of registering softphones and hardphones and intelligently routing calls between VoIP and PSTN. This also includes call-center, IVR, and Voicemail out-of-the-box, and is flexible enough to add custom logic. If our network requires multiple such gateways, we can replicate some of the extra functionality between them and setup call routing in a way that eliminates unneeded voice transfers over the network, thus reducing latency.

The described setup can also be used as a dedicated PSTN system. With this method, you can still use the networking features if your setup consists of more than one node—for configuration data failover or bridging of calls terminated on different nodes.

Advanced usage scenarios for hardware with single or multiple E1 interfaces are usually targeted for signalling. If we take a card with 2 E1 interfaces, cross-connect together all the timeslots except 16 from port 1 to port 2, and then run an application which speaks Q.931 on timeslot 16 of port 1, and transparently translate it to SS7 ISUP on timeslot 16 of port 2, we will have a *signalling converter*. This is used to connect ISDN-only equipment to a SS7-only switch. If we implement SS7 TCAP/MAP, we can create a SMS center out of the same hardware or build IN SCP (Intelligent Network Service Control Point).

Although the E1/T1 connection option is used in the majority of large-scale voice services, you may still need an analog or ISDN BRI line to connect to your server. Digium and others vendors offer analog and ISDN BRI cards which also fit into the Zaptel framework.

3.1 Echo

When interconnecting VoIP and PSTN it is not uncommon to have problems with echo. *Hybrid transition* refers to the situation where the incoming and outgoing signals are passed via a single 2-wire line and separated afterwards, thereby reflecting some of the outgoing signal back. It is also possible for analog phones or headsets to "leak" audio from headphones or speakers to the microphone. Circuit-switched phone networks are very fast and as a result echo is not noticeable. This is because there are two digital-analog conversions on each side and digitized audio is passed in single-byte granularity resulting in low latency. VoIP installations which include voice codecs (adding more overhead) and passing multiple samples in one packet, may introduce enough latency to result in noticable echo.

To eliminate echo, echo cancellation can be added which subtracts the remnants of the outgoing signal from the incoming channel, thus separating them. It is worth mentioning, however, that if in an `A-B` conversation, party A hears an echo, there is nothing you can do on the A side—the problem (unbalanced hybrid, audio leak, broken echo canceller) is on the B side.

3.2 Fax over IP transmission

Faxing over VoIP networks does not work for a number of reasons. First, voice codecs used to reproduce voice data are *lossy*, which confuses faxmodems. Using G.711 can be lossless if you are connected using a digital interface. This is because when used in PSTN DS0 circuits, the unmodified data is passed as the payload to the VoIP protocol. If you do this, however, then echo cancellation will still mangle the signal to the point that modems cannot deal with. As a result, you also need to turn off echo cancellation. Unfortunately, this means the internal modem echo-canceller will not be able to deal with VoIP echo and jitter to the point where it will not work.

There are a number of solutions for this problem. The first is called *T.37—store and forward* protocol. With store and forward, the VoIP gateway on sending end captures the fax and transmits it to the gateway on the receiving side using SMTP. The second method is *T.38*—which tries to emulate *realtime* fax behavior. This is usually more convenient when you send faxes in the middle of the conversation.

4 Zaptel architecture

Most digital interface cards come with one or a combination of interfaces, which together, form a *span*. Data in G.703 E1 stream travels continuously, but cards are usually programmed to signal an interrupt after a predetermined configured amount of data is receiveed in a buffer. In addition, the interrupt is generated when data transmission is finished.

The Zaptel hardware driver provides the following functionality:

- empty data from the device receive buffer, rearrange it channelwise (if needed) and fill the zaptel receive buffer
- call echo cancellation hooks and call `zt_receive(&p->span)`—on the receive path
- call `zt_transmit(&p->span)`, call echo cancellation hooks, empty data of zaptel transmit buffer, rearrange it channelwise and put into device from the transmit buffer, and queue transmission—on the transmit path.

This basic API makes writing drivers very easy. Advanced features can also be implemented too. For example, some E1 cards have the ability to *cross-connect* timeslots without passing the data to the software—useful when both parties are sharing channels of the same span, or for special telephony applications. This feature can be implemented (with some effort) and is supported by current versions of Asterisk.

Clear channels, used for voice data, are passed to userspace unmodified. *Signalling channels*, however, need modification performed by the Zaptel level. ISDN signalling (Q.931 on top of Q.921) requires HDLC framing in the channel, which must be implemented in the kernel. The `ztcfg` tool is used to configure the channel as `D-chan`.

While HDLC framing is done at the kernel-level, Q.931 signalling itself must be done in userspace. Digium offers a library (libpri) for this. This driver was originally used in the zaptel channel driver in asterisk—used now in most ISDN-capable software. SS7 signalling is slightly more difficult as it requires continuous *Fill-In Signal Unit* (FISU) generation which must be placed in the kernel (at the zaptel level) for reliability.

4.1 Code quality issues

Although the zaptel-related hardware driver part seems straighforward, zaptel itself isn't that good. Its 200-kilobyte, 7,000 line single source file includes everything plus the kitchen sink which Asterisk depends heavily on. Due to the continuous flow of data, zaptel devices are often used a as stable source of timing, particularly in the IAX trunking implementation and for playing Music-On-Hold to VoIP clients. To use this feature without Zaptel hardware you need the special `ztdummy` driver which uses RTC and emulates the zaptel timing interface. Also, for reasons we cannot explain, the zaptel kernel module contains a user-space API for conferencing. This module allows the attachment of multiple readers/writers to a particular device node and does all mixing in kernel space. Thus, to enable asterisk conferencing, you also need zaptel hardware or ztdummy. Echo cancellation is selectable and configurable only at compile-time. This is inconvenient when troubleshooting echo problems.

Consistent with every external kernel module that is supposed to work with 2.4 and 2.6 kernels, zaptel contains lots of `#ifdefs` and wrapper macros. It is unclear

if Digium will ever try to push Zaptel to mainline—in its current state we think that is impossible.

4.2 Cost, scalability and reliability

Most telco equipment is overpriced. Although we have found PBXes with an E1 port and 30 customer ports for a reasonable price, the base feature set is often very limited. Each additional feature costs additional money and you still will not receive the level of flexibility provided by open-source software packages. Options for interconnecting PSTN and IP are even more expensive.

Telco equipment is overpriced for a number of reasons—mostly due to reliability and scalability. By building a telco system out of commodity hardware, the only expensive part is the E1 digital interface. Even with this part we are able to keep the cost of single unit low enough to allow $1+1$ (or even $1+1+1spare$) configuration, and the price of hardware will still be much lower. This approach allows us to reach an even higher level of reliability than simply having one telephony switch. This is because we can take units down for maintenance one-by-one.

Combining different existing solutions also reduces some limitations. For example, if the number of VoIP clients in our VoIP-PBX with PSTN connection is so high that asterisk cannot handle the load, we can put a lightweight SIP proxy (OpenSER) in front of it, and all internal VoIP calls will close there.

4.3 Performance issues

There are some inefficiencies in PSTN processing from a performance point of view, which are dictated by the Zaptel architecture. Some cards generate interrupts for each port. For example, with a sample length of 1ms (`ZT_CHUNKSIZE == 8`) there will be 1,000 interrupts per second per port. If we add a large number of ports in a single machine, this number will be multiplied accordingly. There are ways to reduce interrupt load. For example, the card can generate a single interrupt for all its ports. Another way is to use larger samples, but this introduces significant latency and is thus discouraged.

Another zaptel problem is that it creates individual device nodes for every channel it handles. Although with recent kernels, we can easily handle lots of minors, reading from individual channels just does not scale. This can be optimized by feeding all channels via a single device node—but we need to be careful here, because there will be signalling in some timeslots. Also, echo cancellation and DTMF detection can double CPU load. Offloading them to dedicated hardware can save 50% of CPU time.

Better performance can also be achieved by simplifying the hardware driver architecture by eliminating complex processing—echo cancellation or DTMF detection—in the kernel (or interrupt context) by coupling clear channels together before feeding them to userspace. Echo cancellation can be performed on hardware or software—in userspace. However, using software echo cancellation and DTMF detection can be more cost-effective—compare the cost of adding another CPU vs. the cost of hardware EC/DTMF detectors.

However, using more servers with less E1 ports may be wise from a reliability point of view. Modern CPUs have enough processing power to drive 4 E1 interfaces even with a totally unoptimized zaptel stack and userspace. Thus, for large setups we can have any number of 4-port servers connected to a high-speed network. If we are interconnecting with VoIP clients here, we can split the load across the 4-port nodes, and the maximum number of VoIP clients will be no more than 120.

5 Current trends

Until recently, Asterisk dominated the opensource telephony landscape. Zaptel and Asterisk were directed by Digium which sells its own telephony hardware. Recently, however, other players stepped up both on the hardware and software fronts.

Sangoma Technologies, a long time procucer of E1/T1 cards, modified its WANPIPE drivers to support Zaptel. Cronyx Engineering's new drivers package also includes the zaptel protocol module.

There are three issues in the Asterisk universe which resulted in the forking of OpenPBX, later renamed to CallWeaver. Those issues are:

1. Requirement to disclaim all copyrights to Digium on code submission, due to Asterisk dual-licensing and Digium commercial offerings.

2. Because of dual licensing, Asterisk is not dependent on modern software libraries. Instead, it contains embedded (dated) Berkeley DB version 1 for internal storage.

3. Strict Digium control on what changes go into the software.

CallWeaver was forked from Asterisk 1.2 and its development is progressing very rapidly. Less than a year ago they switched from a fragile custom build system to automake, from zaptel timing to POSIX timers, from zaptel conferencing to a userspace mixing engine, from internal DSP functions to Steve Underwood's SpanDSP library, and from awkward db1 to flexible SQLite. CallWeaver has working T.38 support, and is still compatible with zaptel hardware. CallWeaver developers are also trying to fix architectural flaws in Asterisk by allowing proper modularization and changing internal storage from linked lists to hash tables.

Although CallWeaver contains many improvements over Asterisk, it still shares its PBX core, which was designed around some PSTN assumptions. For example, it is assumed that audio data is sampled at 8khz. This is good for pure PSTN applications (or PSTN/VoIP gateways), but in VoIP environments we might want to support other sampling rates and data flows.

FreeSWITCH is designed from the ground up to be more flexible in its core, and uses as many existing libraries and tools as it can. Its development started in January 2006, and although there aren't any official releases at the time of writing this paper, the feature set is already complete—for a softswitch. Unfortunately there is only basic support for PSTN, a native module for Sangoma.

Another software package is Yate, started three years ago. It is written in C++, its source code is an order of magnitude smaller than Asterisk, and it has a cleaner architecture which grants much more flexibility. Yate can use the native WANPIPE interface to drive Sangoma hardware, delivering extremely high performance with high-density Sangoma cards.

6 Conclusion

Running telephony systems with Linux implementaions for the past three years has resulted in the following successful working setups:

1. Pure VoIP IVR and information service for call-center employees using Asterisk.

2. Software load testing of proprietary VoIP equipment using Asterisk.

3. VoIP exchange using Yate.

4. Softswitch with call-center and two E1 PSTN interfaces using Asterisk and Digium E1 equipment.

5. ISDN signalling proxy in Python, using Cronyx E1 equipment.

6. Hybrid PSTN/VoIP telephony network for Saratov State University—multiple gateways using Asterisk and (lately) OpenPBX plus OpenSER on Cronyx E1 equipment.

All implementations were based on i386 and x86_64 hardware platforms and Linux as the operating system kernel. Since these setups were put into operation, we have had no problems with the reliability, stability, or performance of the software we chose. This was a result of careful capacity planning, clustering, and $1+1$ reservations of critical components.

In this paper, we have provided reasons for why building a softswitch or PSTN-connected system from commodity hardware and open-source software may be desirable, and why Linux is a good platform for implementing voice services. However, there are some deficiencies in the current implementations, both in the kernel and in some of the opensource packages, that can potentially result in scalability issues. There are ways to avoid these issues or solve them completely. Our suggestions include improving the Zaptel framework or introducing a new, more efficient framework.

7 References

VOIP Wiki, `http://voip-info.org`

Nathan Willis. *Finding voice codecs for free software.* `http://software.newsforge.com/article.pl?sid=05/09/28/1646243`

Linux Kernel Development

How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It

Greg Kroah-Hartman
SuSE Labs / Novell Inc.
gregkh@suse.de

1 Introduction

The Linux kernel is one of the most popular Open Source development projects, and yet not much attention has been placed on who is doing this development, who is sponsoring this development, and what exactly is being developed. This paper should help explain some of these facts by delving into the kernel changelogs and producing lots of statistics.

This paper will focus on the kernel releases of the past two and 1/3 years, from the 2.6.11 through the 2.6.21 release.

2 Development vs. Stability

In the past, the Linux kernel was split into two different trees, the *development* branch, and the *stable* branch. The development branch was specified by using an odd number for the second release number, while the stable branch used an even number. As an example, the 2.5.32 release was a development release, while the 2.4.24 release is a stable release.

After the 2.6 kernel series was created, the developers decided to change this method of having two different trees. They declared that all 2.6 kernel releases would be considered "stable," no matter how quickly development was happening. These releases would happen every 2 to 3 months and would allow developers to add new features and then stabilize them in time for the next release. This was done in order to allow distributions to be able to decide on a release point easier by always having at least one stable kernel release near a distribution release date.

To help with stability issues while the developers are creating a new kernel version, a `-stable` branch was created that would contain bug fixes and security updates for the past kernel release before the next major release happened.

This is best explained with the diagram shown in Figure 1. The kernel team released the 2.6.19 kernel as a stable release. Then the developers started working on new features and started releasing the `-rc` versions as development kernels so that people could help test and debug the changes. After everyone agreed that the development release was stable enough, it was released as the 2.6.20 kernel.

While the development of new features was happening, the 2.6.19.1, 2.6.19.2, and other stable kernel versions were released, containing bug fixes and security updates.

For this paper, we are going to focus on the main kernel releases, and ignore the `-stable` releases, as they contain a very small number of bugfixes and are not where any development happens.

3 Frequency of release

When the kernel developers first decided on this new development cycle, it was said that a new kernel would be released every 2-3 months, in order to prevent lots of new development from being "backed up." The actual number of days between releases can be seen in Table 1.

It turns out that they were very correct, with the average being 2.6 months between releases.

4 Rate of Change

When modifying the Linux kernel, developers break their changes down into small, individual units of change, called patches. These patches usually do only

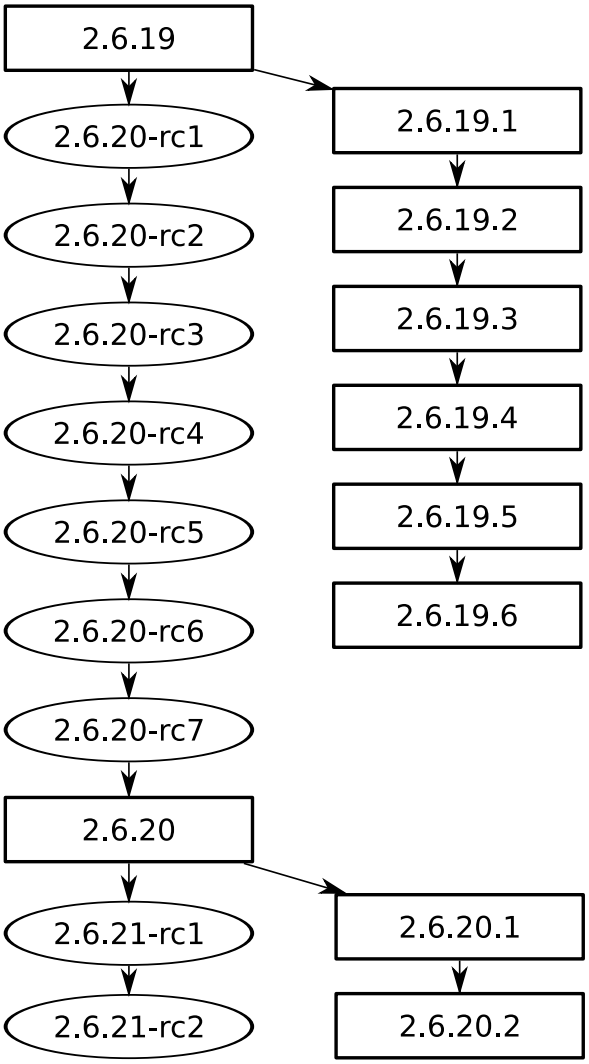


Figure 1: Kernel release cycles

Kernel Version	Days of development
2.6.11	69
2.6.12	108
2.6.13	73
2.6.14	61
2.6.15	68
2.6.16	77
2.6.17	91
2.6.18	95
2.6.19	72
2.6.20	68
2.6.21	81

Table 1: Frequency of kernel releases

Kernel Version	Changes per Release
2.6.11	4,041
2.6.12	5,565
2.6.13	4,174
2.6.14	3,931
2.6.15	5,410
2.6.16	5,734
2.6.17	6,113
2.6.18	6,791
2.6.19	7,073
2.6.20	4,983
2.6.21	5,349

Table 2: Changes per kernel release

one thing to the source tree, and are built on top of each other, modifying the source code by changing, adding, or removing lines of code. At each change point in time, the kernel should be able to be successfully built and operate. By enforcing this kind of discipline, the kernel developers must break their changes down into small logical pieces. The number of individual changes that go into each kernel release is very large, as can be seen in Table 2.

When you compare the number of changes per release, with the length of time for each release, you can determine the number of changes per hour, as can be seen in Table 3.

So, from the 2.6.11 to the 2.6.21 kernel release, a total of 852 days, there were 2.89 patches applied to the kernel

Kernel Version	Changes per Hour
2.6.11	2.44
2.6.12	2.15
2.6.13	2.38
2.6.14	2.69
2.6.15	3.31
2.6.16	3.10
2.6.17	2.80
2.6.18	2.98
2.6.19	4.09
2.6.20	3.05
2.6.21	2.75

Table 3: Changes per hour by kernel release

tree per hour. And that is only the patches that were accepted.

5 Kernel Source Size

The Linux kernel keeps growing in size over time, as more hardware is supported, and new features added. For the following numbers, I count everything in the released Linux source tarball as "source code" even though a small percentage is the scripts used to configure and build the kernel, as well as a minor amount of documentation. This is done because someone creates those files, and are worthy of being mentioned.

The information in Table 4 show the number of files and lines in each kernel version.

Kernel Version	Files	Lines
2.6.11	17,091	6,624,076
2.6.12	17,361	6,777,860
2.6.13	18,091	6,988,800
2.6.14	18,435	7,143,233
2.6.15	18,812	7,290,070
2.6.16	19,252	7,480,062
2.6.17	19,554	7,588,014
2.6.18	20,209	7,752,846
2.6.19	20,937	7,976,221
2.6.20	21,281	8,102,533
2.6.21	21,615	8,246,517

Table 4: Size per kernel release

Over these releases, the kernel team has a very constant growth rate of about 10% per year, a very impressive number given the size of the code tree.

When you combine the number of lines added per release, and compare it to the amount of time per release, you can get some very impressive numbers, as can be seen in Table 5.

Summing up these numbers, it comes to a crazy 85.63 new lines of code being added to the kernel tree every hour for the past 2 1/3 years.

6 Where the Change is Happening

The Linux kernel source tree is highly modular, enabling new drivers and new architectures to be added quite easily. The source code can be broken down into the following categories:

Kernel Version	Lines per Hour
2.6.11	77.6
2.6.12	59.3
2.6.13	120.4
2.6.14	105.5
2.6.15	90.0
2.6.16	102.8
2.6.17	49.4
2.6.18	72.3
2.6.19	129.3
2.6.20	77.4
2.6.21	74.1

Table 5: Lines per hour by kernel release

- **core**: this is the core kernel code, run by everyone and included in all architectures. This code is located in the subdirectories `block/`, `ipc/`, `init/`, `kernel/`, `lib/`, `mm/`, and portions of the `include/` directory.
- **drivers**: these are the drivers for different hardware and virtual devices. This code is located in the subdirectories `crypto/`, `drivers/`, `sound/`, `security/`, and portions of the `include/` directory.
- **architecture**: this is the CPU specific code, where anything that is only for a specific processor lives. This code is located in the `arch/`, and portions of the `include/` directory.
- **network**: this is the code that controls the different networking protocols. It is located in the `net/` directory and the `include/net` subdirectory.
- **filesystems**: this is the code that controls the different filesystems. It is located in the `fs/` directory.
- **miscellaneous**: this is the rest of the kernel source code, including the code needed to build the kernel, and the documentation for various things. It is located in `Documentation/`, `scripts/`, and `usr/` directories.

The breakdown of the 2.6.21 kernel's source tree by the number of different files in the different category is shown in Table 6, while Table 7 shows the breakdown by the number of lines of code.

Category	Files	% of kernel
core	1,371	6%
drivers	6,537	30%
architecture	10,235	47%
network	1,095	5%
filesystems	1,299	6%
miscellaneous	1,068	5%

Table 6: 2.6.21 Kernel size by files

Category	Lines of Code	% of kernel
core	330,637	4%
drivers	4,304,859	52%
architecture	2,127,154	26%
network	506,966	6%
filesystems	702,913	9%
miscellaneous	263,848	3%

Table 7: 2.6.21 Kernel size by lines of code

In the 2.6.21 kernel release, the architecture section of the kernel contains the majority of the different files, but the majority of the different lines of code are by far in the drivers section.

I tried to categorize what portions of the kernel are changing over time, but there did not seem to be a simple way to represent the different sections changing based on kernel versions. Overall, the percentage of change seemed to be evenly spread based on the percentage that the category took up within the overall kernel structure.

7 Who is Doing the Work

The number of different developers who are doing Linux kernel development, and the identifiable companies[1] who are sponsoring this work, have been slowly increasing over the different kernel versions, as can be seen in Table 8.

[1]The identification of the different companies is described in the next section.

Kernel Version	Number of Developers	Number of Companies
2.6.11	479	30
2.6.12	704	38
2.6.13	641	39
2.6.14	632	45
2.6.15	685	49
2.6.16	782	56
2.6.17	787	54
2.6.18	904	60
2.6.19	887	67
2.6.20	730	75
2.6.21	838	68
All	2998	83

Table 8: Number of individual developers and employers

Factoring in the amount of time between each individual kernel releases and the number of developers and employers ends up showing that there really is an increase of the size of the community, as can be shown in Table 9.

Despite this large number of individual developers, there is still a small number who are doing the majority of the work. Over the past two and one half years, the top 10 individual developers have contributed 15 percent of the number of changes and the top 30 developers

Kernel Version	Number of Developers per day	Number of Companies per day
2.6.11	6.94	0.43
2.6.12	6.52	0.35
2.6.13	8.78	0.53
2.6.14	10.36	0.74
2.6.15	10.07	0.72
2.6.16	10.16	0.73
2.6.17	8.65	0.59
2.6.18	9.52	0.63
2.6.19	12.32	0.93
2.6.20	10.74	1.10
2.6.21	10.35	0.84

Table 9: Number of individual developers and employers over time

have contributed 30 percent. The list of individual developers, the number of changes they have contributed, and the percentage of the overall total can be seen in Table 10.

Name	Number of Changes	Percent of Total
Al Viro	1326	2.2%
David S. Miller	1096	1.9%
Adrian Bunk	1091	1.8%
Andrew Morton	991	1.7%
Ralf Baechle	981	1.7%
Andi Kleen	856	1.4%
Russell King	788	1.3%
Takashi Iwai	764	1.3%
Stephen Hemminger	650	1.1%
Neil Brown	626	1.1%
Tejun Heo	606	1.0%
Patrick McHardy	529	0.9%
Randy Dunlap	486	0.8%
Jaroslav Kysela	463	0.8%
Trond Myklebust	445	0.8%
Jean Delvare	436	0.7%
Christoph Hellwig	435	0.7%
Linus Torvalds	433	0.7%
Ingo Molnar	429	0.7%
Jeff Garzik	424	0.7%
David Woodhouse	413	0.7%
Paul Mackerras	411	0.7%
David Brownell	398	0.7%
Jeff Dike	397	0.7%
Ben Dooks	392	0.7%
Greg Kroah-Hartman	388	0.7%
Herbert Xu	376	0.6%
Dave Jones	371	0.6%
Ben Herrenschmidt	365	0.6%
Mauro Chehab	365	0.6%

Table 10: Individual Kernel contributors

8 Who is Sponsoring the Work

Despite the broad use of the Linux kernel in a wide range of different types of devices, and reliance of it by a number of different companies, the number of individual companies that help sponsor the development of the Linux kernel remains quite small as can be seen by the list of different companies for each kernel version in Table 8.

The identification of the different companies was deduced through the use of company email addresses and the known sponsoring of some developers. It is possible that a small number of different companies were missed, however based on the analysis of the top contributors of the kernel, the majority of the contributions are attributed in this paper.

The large majority of contributions still come from individual contributors, either because they are students, they are contributing on their own time, or their employers are not allowing them to use their company email addresses for their kernel development efforts. As seen in Table 11 almost half of the contributions are done by these individuals.

Company Name	Number of Changes	Percent of Total
Unknown	27976	47.3%
Red Hat	6106	10.3%
Novell	5923	10.0%
Linux Foundation	4843	8.2%
IBM	3991	6.7%
Intel	2244	3.8%
SGI	1353	2.3%
NetApp	636	1.1%
Freescale	454	0.8%
linutronix	370	0.6%
HP	360	0.6%
Harvard	345	0.6%
SteelEye	333	0.6%
Oracle	319	0.5%
Conectiva	296	0.5%
MontaVista	291	0.5%
Broadcom	285	0.5%
Fujitsu	266	0.4%
Veritas	219	0.4%
QLogic	218	0.4%
Snapgear	214	0.4%
Emulex	147	0.2%
LSI Logic	130	0.2%
SANPeople	124	0.2%
Qumranet	106	0.2%
Atmel	91	0.2%
Toshiba	90	0.2%
Samsung	82	0.1%
Renesas Technology	81	0.1%
VMWare	78	0.1%

Table 11: Company Kernel Contributions

9 Conclusion

The Linux kernel is one of the largest and most successful open source projects that has ever come about. The

huge rate of change and number of individual contributors show that it has a vibrant and active community, constantly causing the evolution of the kernel to survive the number of different environments it is used in. However, despite the large number of individual contributors, the sponsorship of these developers seem to be consolidated in a small number of individual companies. It will be interesting to see if, over time, the companies that rely on the success of the Linux kernel will start to sponsor the direct development of the project, to help ensure that it remains valuable to those companies.

10 Thanks

The author would like to thank the thousands of individual kernel contributors, without them, papers like this would not be interesting to anyone.

I would also like to thank Jonathan Corbet, whose `gitdm` tool were used to create a large number of these different statistics. Without his help, this paper would have taken even longer to write, and not been as informative.

11 Resources

The information for this paper was retrieved directly from the Linux kernel releases as found at the `kernel.org` web site and from the git kernel repository. Some of the logs from the git repository were cleaned up by hand due to email addresses changing over time, and minor typos in authorship information. A spreadsheet was used to compute a number of the statistics. All of the logs, scripts, and spreadsheet can be found at `http://www.kernel.org/pub/linux/kernel/people/gregkh/kernel_history/`

Implementing Democracy

a large scale cross-platform desktop application

Christopher James Lahey
Participatory Culture Foundation
clahey@clahey.net

Abstract

Democracy is a cross-platform video podcast client. It integrates a large number of functions, including searching, downloading, and playing videos. Thus, it is a large-scale application integrating a number of software libraries, including a browser, a movie player, a bittorrent client, and an RSS reader.

The paper and talk will discuss a number of techniques used, including using PyRex to link from python to C libraries, using a web browser and a templating system to build the user interface for cross-platform desktop software (including a different web browser on each platform), and our object store used to keep track of everything in our application, store our state to disk, and bring updates to the UI.

1 Internet video

Internet video is becoming an important part of modern culture, currently through video blogs, video podcasts, and YouTube. Video podcasting gives everyone the ability to decide what they want to make available, but the spread of such systems as YouTube and Google Video suggest that large corporations will have a lot to say in the future of internet video.

The most popular use of internet video right now is YouTube. YouTube videos are popping up all over the place. Unfortunately, this gives one company a lot of power over content. It lets them take down whatever they find inconvenient or easily block certain content from reaching certain people.

Video podcasts are RSS feeds with links to videos. Podcasting allows the publisher to put whatever videos he wants on his personal webspace. Podcasting clients download these videos for display on different devices. This gets around the problem of one company controlling everything. That is, except for the fact that the most prominent podcast client is iTunes and it's used for downloading to iPods. Once again, the company has the ability to censor.

Some folks in Worcester, Massachusetts saw this as a problem and so sought funding and formed the nonprofit Participatory Culture Foundation. The goal of the Participatory Culture Foundation is to make sure that everyone has a voice and that no one need be censored. We are approaching this from a number of different angles. We have a project writing tutorials for people that want to make and publish internet video. We are in planning for a server project to let people post their video podcasts. And most importantly, we write the Democracy player.

The reason the player is so important to us is that we want to make sure that publishing and viewing are independent. If they aren't, then there are two types of lock-in. Firstly, if a user wants to see a particular video, they're forced to use the particular publisher's viewing software. Secondly, once a user starts using a particular publisher's viewing software, that publisher gains control over what the viewer can see. These two could easily join together in a feedback loop that leads to a monopoly situation.

However, to separate publishing and viewing, we need a standard for communication. RSS fills this role perfectly. In fact, it's already in use for this purpose. The role we want Democracy to fill is that of a good player that encourages viewers to use RSS. Well, we also just want it to be a great video player.

2 Democracy

Democracy's main job is to download and play videos from RSS feeds. We also decided to make it able to be

the heart of your video experience. Thus it plays local videos, searches for videos on the internet, and handles videos downloaded in your web browser.

To do all this we integrated a number of other tools. We included other python projects wholesale, like feedparser and BitTorrent. We link to a number of standard libraries through either standard python interfaces or through the use of Pyrex as a glue language.

For widest adoption, we decided it was important for Democracy to be cross-platform. Windows and Mac would get us the most users, but Linux is important to us since we create free software. So far two of our new developers (myself included) have come from Linux-land and one from OSX-land.

3 Major objects in Democracy

There are two major object types that we deal with: Feeds and Items.

Feeds tend to be RSS feeds, but they can also be scrapes of HTML pages, watches on local directories, and other things. Since we don't know at feed creation time whether a URL will return an RSS feed or an HTML page, we create a Feed object which is mainly a proxy to a FeedImpl object that can be created later. python makes this proxy action almost completely transparent. We implement a `__getattr__` handler which gets called for methods and data that aren't defined in the Feed object. In this handler, we simply return the corresponding method or data for the FeedImpl object. We use this trick in a couple of other similar proxy situations in Democracy.

Items are individual entries in an RSS feed. Most of them represent a video that democracy could potentially download. You can also use democracy to download either directories of multiple videos or non-video files. They can be either available, downloading, or downloaded. We also have FileItems which are used to designate local files that don't have a corresponding URL. These can either be the children videos of a directory we download, or files found on the local disk. We have special feeds that monitor a local directory and create any found files. You can also pass Democracy a filename and it will create an item for that video.

We've spent a lot of time tweaking the behavior of all of these objects. One of the things we've discovered is that the more features that we have, the harder new features are to implement. Anyone who has any experience at all shouldn't be surprised to hear this, but it's amazing the difference between hearing about it in books and getting specific examples in your work.

4 Object Store

To keep track of everything that is happening in our application, we have a collection of objects. Every important object has a global ID. This includes all feeds and items, as well as playlists and channel guides.

However, we need fast access to different sets of objects. We need a list of all items in a particular feed, for example. To implement this, we have a system of views into the database. Each view acts like another database and can thus have subviews. We have a number of different view types. The first is *filters*, which are subsets of the database. The second is *maps*, which create a new object for each member of the original database. The third is *indexes*, which create a whole bunch of subdatabases and put each item in one of the subdatabases. Finally we have *sorts*, though these are redundant, as the other view types can be sorted as well.

The other important part of the object database is that you can monitor a view for changes. We send signals on new objects being added or removed. Each object has a signalChange function which signals that the data in that object has changed. You monitor this by watching for changes on a database view.

This in-memory object database has worked quite well for us. We have a list of all objects that we care about, while still being able to have lists of the objects that we care about right now. An example of a use of views is that each feed doesn't keep a list of its items. It just has a view into the database and as the items get created, the feed gets that item added to its item list. However, the biggest use of views is in our cross-platform UI.

5 Cross Platform UI

To implement the cross-platform user interface, we use HTML with CSS. We have two main HTML areas and platform-specific menus and buttons. The HTML is generated automatically based on the objects in our object database.

We start with XML templates that are compiled into python code that in turn generates HTML. You can pass in plain XHTML and it will work. To start with, we had a series of key-value pairs that were set in python code and could then be accessed within the templates. That proved to be a pain of having to change the python every time we needed to change a referenced key, so we switched to simply allowing the python to be embedded directly in the XML. This is safe, since we provide all the XML and don't take any from the outside world, and it's much more maintainable.

The template system can do a number of different things with the results of the python. It can embed the result directly in the HTML. It can encode as a string and then embed the result in the HTML.

Slightly more interestingly, it can hide a chunk of HTML based on the return value. It can update a section of the template whenever a view changes. The most interesting part though is that it can repeat a section of HTML for every object in a view.

`repeatForView` takes a chunk of template and repeats it for every object in a view. When an object is added to or removed from the view, it adds or removes the corresponding HTML. When an object changes, it recalculates the HTML for that object.

Some of our team members are not entirely happy with HTML as our solution. It means working within the system that the browser gives us. It also means supporting both OSX's webkit and mozilla with our code (we use `gtkmozembed` on Linux and `xul` on Windows.) Finally, it sticks us with `xul` on Windows. We originally tried using the Windows framework by hand. We decided this was just too much work from python. When that didn't work, we tried using gtk and embedding mozilla, but found that `gtkmozembed` doesn't work on Windows. Finally we switched to `xul`, but `xul` is much harder to code to than either OSX or gtk. We may switch to using gtk+ on Windows, and to support that, we would switch to using some other rendering system, perhaps our own XML language that maps to cairo on gtk+ and something else on OSX.

I personally would prefer to stick with HTML plus CSS. It gives us a wide range of developers who know our rendering model. It gives us a bunch of free code to do the rendering. The only problem is getting one of those sets of code to work on Windows.

6 LiveStorage

To save our database to disk, we originally just pickled the object database to disk. Python pickle is a library that takes a python object and encodes all the data in it to an on-disk format. The same library will then decode that data and create the corresponding objects in memory again. It handles links to other objects including reference loops and it handles all the standard string and number data types.

This worked as long as we didn't change the in-memory representation of any of our objects. We worked around a number of different issues, but in the end we decided to remove some classes, and pickle throws an exception if it has an object on disk that doesn't have a corresponding class in memory.

The next step was to add a system that copied the data into python dictionaries and then pickled that created object. To do this, we created a schema object which describes what sort of data gets stored. This makes removing a field trivial. The system that copies the data out of the python dictionaries at load time simply ignores any fields not listed in the schema.

Adding fields is a bit more complicated, but to solve this, we store a version number in the database. Every time we change the schema, we increment this version number. At load time, the system compares the version in the database to the version in the running application and runs a series of upgrades on the data. These upgrades happen on the dictionary version of the objects and thus involve no running code. They can also remove or add objects, which allows us to remove old classes that aren't necessary and add objects that are.

Our next problem with data storage was that it was super slow to save the database. The larger the database got, the slower it was, to the tune of 45 seconds on large data sets, and we want to save regularly so that the user doesn't lose any data.

To solve this, we decided to save each object individually. Unfortunately, the objects referred to one another. Pickle handles this just fine when you ask it to pickle all the objects at once, but it isn't able to do that when you want to pickle just a single object (in fact, it will basically pickle every object related to the one you requested and then at load time will not connect the objects saved in different pickle runs.) The biggest example of this was that each feed kept a list of the items in that feed.

So the first step was to make the objects not refer to each other. For the most part we wanted to do this by just replacing references to other objects with their database ID. This works, but we also decided that we didn't want to keep redundant data. For example, a simple replacement of references with IDs in the database would lead to a feed having a list of items in that feed and the items each having the ID of their feed. Luckily, our in-memory database already had filters. We just made the feeds not store their children, and instead the list of children is simply a database filter.

After this, we replaced the single pickle with a Berkeley database storing the list of objects. We still had to worry about keeping changes in sync. For example, when first creating a feed, you need to make sure that all of the items are saved as well. To solve this, we simply stored the list of changed items and did the actual save to the database in a timeout loop. We used a transaction, and since the old database save happened in a timeout loop as well, we have the exact same semantics for syncing of different objects.

This worked great for a good while. We even used the schema upgrade functions to do things other than database upgrades, such as to automatically work around bugs in old versions. In fact we had no problem with this on Linux or Windows, but on Mac OSX, we got frequent reports of database errors on load and many people lost their data. We tried reproducing the error to debug it and we asked about the issue on Berkeley DB's usenet groups (where we'd gotten useful information before,) but there was no response. From there we decided to switch to `sqlite`. We're still using it to just store a list of pickled objects, but it's working fairly well for us.

The next step we'd like to take is not to have the entire database in memory at all times. Having it in memory increases our memory usage and limits the number of feeds a user can monitor. We'd like to change to using a more standard relational database approach to storing our data. This would, however, completely change how we access our data. We've decided that the size of this change means we should wait until after 1.0 to make this change.

There are some other major obstacles to making this change other than the number of pieces of code that would have to change. The first is that we use change notification extensively. An object changes and people interested in that object are notified. Similarly, objects added to or removed are signalled. To get around this, we would need either a relational database that does change notifications of this sort, or we would need to add a change notification layer on top of the database. Currently these notifications happen based on the different filters in our database, so we would need to duplicate the SQL searches that we do as monitoring code to know who needs to be signalled. For this reason, we're hoping that we find a relational database that will handle live sql searches and send notification of changes, additions, and removals.

The second obstacle is less of a problem with the change and more a reason that it won't help. Specifically, we touch most of the database at load time anyway. It would mean that we could start the user interface having loaded less data, but we queue an update of every RSS feed at load time. To run this update, we need to load all the items in that feed so we can compare them to the items we download (so that we don't create duplicate items.) At that point we could unload all that data.

7 BitTorrent

We act as a bittorrent client as well. This can either be by loading a BitTorrent file by hand or by including it in an RSS player. As a user, I found it very pleasant to have things just download. In fact, at first, I didn't realize that I was using a BitTorrent client. I think this can help increase usage of BitTorrent since people won't be intimidated by technology if they don't know they're using it. Another good example of this is the downloader used for World of Warcraft.

Unfortunately, the primary BitTorrent source code has become non-free software. For this reason, we eventually switched to using BitTornado. Unfortunately BitTornado introduced a number of bugs, and we decided that for us, fixing those bugs would be harder than recreating the new features that BitTornado supplied. We still don't have all the features that we want, but in switching back to the old BitTorrent code, we've got a codebase that tends to work quite well.

Looking into the future, we'd love to see a free software BitTorrent client take off. Post 1.0, we're considering adding a bunch of new BitTorrent features and protocol improvements to the code base we're currently using. Our goal would certainly be to maintain the code

as a separate project so we don't waste our time and so that the free software community gets a good BitTorrent client. Of course, this would depend on other developers, but we will see what happens.

8 FeedParser

For parsing RSS feeds, we've had a huge amount of success with `feedparser.py`. It's a feed parser designed to be used either separately or as a library. It parses the RSS feeds and gives them to us as useful data structures. So far the library has just worked for us in almost every situation.

The only thing we've had trouble with has been that feedparser derives a new class based on python dictionaries. It does this so that it can treat a number of different keys as the same key. For instance, `url` and `location` are treated as the same key, so that if you set either one of them, `parser_dict["url"]` will give the value. Unfortunately, this aggregation is done at read time instead of write time. This makes the dictionary a bit slower to use, but more importantly, it's meant that the == operator doesn't have the behavior that you might naively expect. We've had to write a fairly complicated replacement for it which is probably much slower than == on two dictionaries. We may change this behavior going forward to change the keys when writing to the dictionaries instead, but I will resist it until we have profiling data that shows that it slows things down.

9 Unicode

python currently has two classes to represent strings. The first is `str`, which is a list of bytes, and the second is `unicode`, which is a list of characters. These two classes automatically convert back and forth as needed, but this conversion can both get the wrong value and can cause exceptions. This usually happens because the automatic conversion isn't quite the conversion you were expecting. It actually assumes ASCII on many machines and just throws an exception when you mix a `str` object and a `unicode` object with characters greater than 127.

Unfortunately, these sorts of bugs rarely show up for an English speaker because most text is ASCII and thus converts correctly. Therefore the bugs can be hard to reproduce and since can happen all over the place.

A reasonable solution might be to use `unicode` everywhere. This was our general policy for a long time, but there were exceptions. The first was developer forgetfulness. When you write a literal string in python, unless you specify that it's `unicode`, it creates a `str` object. The second exception is that there are certain python classes that only work with `str` objects, such as `cStringIO`. For these classes we would convert into `str`s of a certain encoding, but we would sometimes forget and we would do multiple conversion steps in some cases. Thirdly, there are OS differences. Specifically, filenames are actually different classes on different OSes. When you do a `listdir` on Windows, you get `unicode` objects in the returned list, but on Linux and OSX, you get `str` objects.

Our recently introduced policy is twofold. First is that you can use different types of objects in different places. We define three object types. There's `str`, there's `unicode`, and there's `filenameType`. `filenameType` is defined in the platform-specific code, so it is different on the different platforms.

In the platform-specific code, we also provide conversion functions between the different types. There's `unicodeToFilename` and `filenameToUnicode` which do the obvious conversion. They do not do reversible conversions, but instead provide a conversion that a user would be happy to see. We also have `makeURLSafe`, and because we need to undo that change, `unmakeURLSafe`. `unmakeURLSafe(makeURLSafe(obj)) == obj` if `obj` is a `filenameType`. We will see if these conversion functions are sufficient or if we need to make more functions like this.

The second part of our policy is that we enforce the types passed to and returned from many functions. We've introduced functions that take a passed-in object and check that they're of the right type and we've introduced decorators that check the return value of the method. In both cases, an exception is thrown if an object is of the wrong type. This means that we see many bugs much sooner than if we just waited for them to be found by people using other languages.

There has been a period of transition to these new policies, since all the exposed bugs have to be fixed. We're still going through this transition phase, but it's going well so far.

10 More info

You can learn more about the democracy project at `getdemocracy.com` and more about the Participatory Culture Foundation at `participatoryculture.org`.

Extreme High Performance Computing or Why Microkernels Suck

Christoph Lameter
sgi
clameter@sgi.com

Abstract

One often wonders how well Linux scales. We frequently get suggestions that Linux cannot scale because it is a monolithic operating system kernel. However, microkernels have never scaled well and Linux has been scaled up to support thousands of processors, terabytes of memory and hundreds of petabytes of disk storage which is the hardware limit these days. Some of the techniques used to make Linux scale were per cpu areas, per node structures, lock splitting, cache line optimizations, memory allocation control, scheduler optimizations and various other approaches. These required significant detail work on the code but no change in the general architecture of Linux.

The presentation will give an overview of why Linux scales and shows the hurdles microkernels would have to overcome in order to do the same. The presentation will assume a basic understanding of how operating systems work and familiarity with what functions a kernel performs.

1 Introduction

Monolithic kernels are in wide use today. One wonders though how far a monolithic kernel architecture can be scaled, given the complexity that would have to be managed to keep the operating system working reliably. We ourselves were initially skeptical that we could go any further when we were first able to run our Linux kernel with 512 processors because we encountered a series of scalability problems that were due to the way the operating system handled the unusually high amounts of processes and processors. However, it was possible to address the scalability bottlenecks with some work using a variety of synchronization methods provided by the operating system and we were then surprised to only encounter minimal problems when we later doubled the number of processors to 1024. At that point the primary difficulties seemed to shift to other areas having more to do with the limitation of the hardware and firmware. We were then able to further double the processor count to two thousand and finally four thousand processors and we were still encountering only minor problems that were easily addressed. We expect to be able to handle 16k processors in the near future.

As the number of processors grew so did the amount of memory. In early 2007, machines are deployed with 8 terabytes of main memory. Such a system with a huge amount of memory and a large set of processors creates the high performance capabilities in a traditional Unix environment that allows for the running of traditional applications, avoiding major efforts to redesign the basic logic of the software. Competing technologies, such as compute clusters, cannot offer such an environment. Clusters consist of many nodes that run their own operating systems whereas scaling up a monolithic operating system has a single address space and a single operating system. The challenge in clustered environments is to redesign the applications so that processing can be done concurrently on nodes that only communicate via a network. A large monolithic operating system with lots of processors and memory is easier to handle since processes can share memory which makes synchronization via Unix shared memory possible and data exchange simple.

Large scale operating systems are typically based on Non-Uniform Memory Architecuture (NUMA) technology. Some memory is nearer to a processor than other memory that may be more distant and more costly to access. Memory locality in such a system determines the overall performance of the applications. The operating system has a role in that context of providing heuristics in order to place the memory in such a way that memory latencies are reduced as much as possible. These have to be heuristics because the operating system cannot know how an application will access allocated memory in the future. The access patterns of the application should ide-

ally determine the placement of data but the operating system has no way of predicting application behavior. A new memory control API was therefore added to the operating system so that applications can set up memory allocation policies to guide the operating system in allocating memory for the application. The notion of memory allocation control is not standardized and so most contemporary programming languages have no ability to manage data locality on their own.[1] Libraries need to be provided that allow the application to provide information to the operating system about desirable memory allocation strategies.

One idea that we keep encountering in discussions of these large scale systems is that Micro-kernels should allow us to handle the scalability issues in a better way and that they may actually allow a better designed system that is easier to scale. It was suggested that a microkernel design is essential to manage the complexity of the operating systems and ensure its reliable operation. We will evaluate that claim in the following sections.

2 Micro vs. Monolithic Kernel

Microkernels allow the use of the context control primitives of the processor to isolate the various components of the operating system. This allows a fine grained design of the operating system with natural APIs at the boundaries of the subsystems. However, separate address spaces require context switches at the boundaries which may create a significant overhead for the processors. Thus many micro kernels are compromises between speed and the initially envisioned fine grained structure (a hybrid approach). To some extent that problem can be overcome by developing a very small low level kernel that fits into the processor cache (for example L4) [11], but then we no longer have an easily programmable and maintainable operating system kernel. A monolithic kernel usually has a single address space and all kernel components are able to access memory without restriction.

2.1 IPC vs. function call

Context switches have to be performed in order to isolate the components of a microkernel. Thus communication between different components must be controlled through an Inter Process Communication mechanism that incurs similar overhead to a system call in monolithic kernel. Typically microkernels use message queues to communicate between different components. In order to communicate between two components of a microkernel the following steps have to be performed:

1. The originating thread in the context of the originating component must format and place the request (or requests) in a message queue.
2. The originating thread must somehow notify the destination component that a message has arrived. Either interrupts (or some other form of signaling) are used or the destination component must be polling its message queue.
3. It may be necessary for the originating thread to perform a context switch if there are not enough processors around to continually run all threads (which is common).
4. The destination component must now access the message queue and interpret the message and then perform the requested action. Then we potentially have to redo the 4 steps in order to return the result of the request to the originating component.

A monolithic operating system typically uses function calls to transfer control between subsystems that run in the same operating system context:

1. Place arguments in processor registers (done by the compiler).
2. Call the subroutine.
3. Subroutine accesses registers to interpret the request (done by compiler).
4. Subroutine returns the result in another register.

From the description above it is already evident that the monolithic operating system can rely on much lower level processor components than the microkernel and is well supported by existing languages used to code for operating system kernels. The microkernel has to

[1] There are some encouraging developments in this area with Unified Parallel C supporting locality information in the language itself. See Tarek, [2].

manipulate message queues which are higher level constructs and—unlike registers—cannot be directly modified and handled by the processor.[2]

In a large NUMA system an even more troubling issue arises: while a function call uses barely any memory on its own (apart from the stack), a microkernel must place the message into queues. That queue must have a memory address. The queue location needs to be carefully chosen in order to make the data accessible in a fast way to the other operating system component involved in the message transfer. The complexity of making the determination where to allocate the message queue will typically be higher than the message handling overhead itself since such a determination will involve consulting system tables to figure out memory latencies. If the memory latencies are handled by another component of the microkernel then queuing a message may require first queuing a message to another subsystem. One may avoid the complexity of memory placement in small configurations with just a few memory nodes but in a very large system with hundreds of nodes the distances are a significant performance issue. There is only a small fraction of memory local to each processor and so it is highly likely that a simple minded approach will cause excessive latencies.

It seems that some parts of the management of memory latency knowledge cannot be handled by a subsystem but each subsystem of the microkernel must include the necessary logic to perform some form of advantageous data placement. It seems therefore that each microkernel component must at least contain pieces of a memory allocator in order to support large scale memory architectures.

2.2 Isolation vs. integration of operating system components

The fundamental idea of a microkernel is to isolate components whereas the monolithic kernel is integrating all the separate subsystems into one common process environment. The argument in favor of a microkernel is that it allows a system to be fail safe since a failure may be isolated into one system component.

[2]There have been attempts to develop processors that handle message queues but no commercially viable solution exists. In contemporary High Performance Computing messages based interfaces are common for inter process communication between applications running on different machines.

However, the isolation will introduce additional complexities. Operating systems usually service applications running on top of them. The operating system must track the state of the application. A failure of one key component typically includes also the loss of relevant state information about the application. Some microkernel components that track memory use and open files may be so essential to the application that the application must terminate if either of these components fails. If one wanted to make a system fail safe in a microkernel environment then additional measures, such as check pointing, may have to be taken in order to guarantee that the application can continue. However, the isolation of the operating system state into different modules will make it difficult to track the overall system state that needs to be preserved in order for check pointing to work. The state information is likely dispersed among various separate operating system components.

The integration into a single operating system process of a monolithic operating system enables access to all state information that the operating system keeps on a certain application. This seems to be a basic requirement in order to enable fail safe mechanisms like check pointing. Isolation of operating system components may actually make reliable systems more difficult to realize.

Performance is also a major consideration in favor of integration. Isolation creates barriers for accessing operating system state information that may be required in order for the operating system to complete a certain task. Integration allows access to all state information by any operating system component.

Monolithic kernels today are complex. An operating system may contain millions of lines of code (Linux currently has 1.4 million lines). There is the impossibility of auditing all that code in order to be sure that the operating system stays secure. An approach that isolates operating system components is certainly beneficial to insure secure behavior of the components. In a monolithic kernel methods have to be developed to audit the kernel automatically or manually by review. In the Linux kernel we have the example of a community review network that keeps verifying large sections of the kernel. Problems can be spotted early to secure the integrity of the kernel. However, such a process may be only possible for community based code development where a large number of developers is available. A single company may not have the resources to keep up the necessary ongoing review of source code.

2.3 Modularization

Modularization is a mechanism to isolate operating system components that may also occur in monolithic kernels. In microkernels this is the prime paradigm and modularization results in modules with a separate process state, a separate executable and separate source code. Each component can be separately maintained and built. The strong modularization usually does not work well for monolithic operating systems. Any part of the operating system may refer to information from another part. Mutual dependencies exist between many of the components of the operating system. Therefore the operating system kernel has to be built as a whole. Separate executable portions can only be created by restricting the operating system state information accessible by the separated out module.

What has been done in monolithic operating systems is the implementation of a series of weaker modes of modularization at a variety of levels.

2.3.1 Source code modularization

There are a number of ways to modularize source code. Code is typically arranged into a directory structure that in itself imposes some form of modularization. Each C source code piece can also be seen as a modular unit. The validity of identifiers can be restricted to one source module only (for example through the **static** attribute in C). The scoping rules of the compiler may be used to control access to variables. Hidden variables are still reachable within the process context of the kernel but there is no way to easily reach these memory locations via a statement in C.

Header files are another typical use of modularization in the kernel. Header files allow the exposing of a controlled API to the rest of the kernel. The other components must use that API in order to use the exported services. In many ways this is similar to what the strict isolation in a microkernel would provide. However, since there is no limitation to message passing methods, more efficient means of providing functionality may be used. For example it is typical to define macros for performance sensitive operations in order to avoid function calls. Another method is the use of in line functions that also avoid function calls. Monolithic kernels have more flexible ways of modularization. It is not that the general idea of modularization is rejected, it is just that the microkernels carry the modularization approach too far. The rigidity of microkernel design limits the flexibility to design APIs that provide the needed performance.

2.3.2 Loadable operating system modules

One way for monolithic operating systems to provide modularity is through loadable operating system modules. This is possible by exposing a binary kernel API. The loaded modules must conform to that API and the kernel will have to maintain compatibility to that API. The loaded modules run in the process context of the kernel but have only access to the rest of the kernel through the exported API.

The problem with these approaches is that the API becomes outdated over time. Both the kernel and the operating system modules must keep compatibility to the binary API. Over time updated APIs will invariably become available and then both components may have to be able to handle different releases of the APIs. Over time the complexity of API—and the necessary workarounds to handle old versions of the API—increases.

Some open source operating systems (most notably Linux) have decided to not support stable APIs. Instead each kernel version exports its own API. The API fluctuates from kernel release to kernel release. The idea of a stable binary API was essentially abandoned. These approaches work because of source code availability. Having no stable API avoids the work of maintaining backward compatibility to previous APIs. Changes to the API are easy since no guarantee of stability has been given in the first place. If all the source code of the operating system and of the loadable modules is available then changes to the kernel APIs can be made in one pass through all the different components of the kernel. This will work as long as the API stays consistent within the source of one kernel release but it imposed a mandate to change the whole kernel on those submitting changes to the kernel.

2.3.3 Loadable drivers

A loadable driver is simply a particular instance of a loadable operating system module. The need to support

loadable device drivers is higher though than the need to support loadable components of the operating system in general since operating systems have to support a large quantity of devices that may not be in use in a particular machine on which the operating system is currently running. Having loadable device drivers cuts down significantly in terms of the size of the executable of the operating system.

Loadable device drivers are supported by most operating systems. Device drivers are a common source of failures since drivers are frequently written by third parties with limited knowledge about the operating system. However, even here different paradigms exist. In the Linux community, third party writing of drivers is encouraged but then community review and integration into the kernel source itself is suggested. This usually means an extended review process in which the third party device driver is verified and updated to satisfy all the requirements of the kernel itself. Such a review process increases the reliability and stability of device drivers and reduces the failure rate of device drivers.

Another solution to the frequent failure of device drivers is to provide a separate execution context for these device drivers (as done in some versions of the Microsoft Windows operating system). That way failures of device drivers cannot impact the rest of the operating system. In essence this is the same approach as suggested by proponents of microkernels. Again these concepts are used in a restricted context. Having a special operating system subsystem that creates a distinct context for device drivers is expensive. The operating system already provides such contexts for user space. The logical path here would be to have device drivers that run in user space thus avoiding the need to maintain a process context for device drivers.

3 Techniques Used to Scale Monolithic Kernels

Proper serialization is needed in order for monolithic operating systems—such as Linux—to run on large processor counts. Access to core memory structures needs to be serialized in such a way that a large number of processors can access and modify the data as needed. Cache lines are the units in which a processor handles data. Cache lines that are read-only are particularly important for performance since these cache lines can be shared. A cache line that is written has first to be removed from all processors that have a copy of that cache line. It is therefore desirable to have data structures that are not frequently written to.

The methods that were used to make Linux scale are discussed in the following sections. They are basically a variety of serialization methods. As the system was scaled up to higher and higher processor counts a variety of experiments were performed to see how each data structure needed to be redesigned and what type of serialization would need to be employed in order to reach the highest performance. Development of higher scalability is an evolutionary approach that involves various attempts to address the performance issues that were discovered during testing.

3.1 Serialization

The Linux kernel has two basic ways of locking. *Semaphores* are sleeping locks that require a user process context. A process will go to sleep and the scheduler will run other processes if the sleeping lock has already been taken by another process. Spinlocks are used if there is no process context. Without the process context we can only repeatedly check if the lock has been released. A spinlock may create high processor usage because the processor is busy continually checking for a lock to be released. Spinlocks are only used for locks that have to be held briefly.

Both variants of locking come in a straight lock/unlock and a reader/writer lock version. Reader/writer locks allows multiple readers and only one writer. Lock/unlock is used for simple exclusion.

3.2 Coarse vs. fine grained locking

The Linux kernel first became capable of supporting multiprocessing by using a single large lock, the Big Kernel Lock (BKL).[3] Over time, coarse grained locks were gradually replaced with finer grained locks. The evolution of the kernel was determined by a continual stream of enhancements by various contributors to address performance limitations that were encountered when running common computing loads. For example

[3] And the BKL still exists for some limited purposes. For a theoretical discussion of such a kernel, see Chapter 9, "Master-Slave Kernels," in [12].

the page cache was initially protected by a single global lock that covered every page. Later these locks did become more fine grained. Locks were moved to the process level and later to sections of the address space. These measures gradually increased performance and allowed Linux to scale better and better on successively larger hardware configurations. Thereby it became possible to support more memory and more processors.[4]

A series of alternate locking mechanisms were proposed. In addition to the four types of locking mentioned above, new locking schemes for special situations were developed. For example **seq_locks** emerged as a solution to the problem of reading a series of values to determine system time. **seq_locks** do not block, they simply repeat a critical section until sequence counters taken at the beginning and end of the critical section indicate that the result was consistent.[5]

Creativity to develop finer-grained locking that would reach higher performance was targeted to specific areas of the kernel that were particularly performance sensitive. In some areas locking was avoided in favor of lockless approaches using atomic operations and Read-Copy-Update (RCU) based techniques. The evolution of new locking approaches is by no means complete. In the area of page cache locking there exists—for example—a project to develop ways to do page cache accesses and updates locklessly via a combination of RCU and atomic operations [9].

The introduction of new locking methods involves various tradeoffs. Finer grained locking requires more locks and more complex code to handle the locks the right way. Multiple locks may be interacting in complex ways in order to ensure that a data structure maintains its consistency. The justification of complex locking schemes became gradually easier as processor speeds increased and memory speeds could not keep up. Processors became able to handle complex locking protocols using locking information that is mostly in the processor caches to negotiate access to data in memory that is relatively expensive to access.

3.3 Per cpu structures

Access to data via locking is expensive. It is therefore useful to have data areas that do not require locking. One such natural area is data that can only be accessed by a single processor. If no other processors use the data then no locking is necessary. This means that a thread of execution needs to be bound to one single processor as long as the per cpu data is used. The process can only be moved to another processor if no per cpu data is used.

Linux has the ability to switch the rescheduling a kernel thread off by disabling preemption. A counter of the number of preemptions taken is kept to allow nested access to multiple per cpu data structures. The execution thread will only be rescheduled to run on other processors if the preemption counter is zero.

Each processor usually has its own memory cache hierarchy. If a cache line needs to be written then it needs to be first cleared from the caches of all other processors. Thus dirtying a cache line is an expensive operation if copies of a cache line exist in the caches of other processors. The cost of dirtying a cache line increases with the number of processors in the system and with the latency to reach memory.

Per cpu data has performance advantages because it is only accessed by a single cpu. There will be no need to clear cache lines on other processors. Memory for per cpu areas is typically set up early in the bootstrap process of the kernel. At that point it can be placed in memory that has the shortest latency for the processor the memory is attached to. Thus memory accesses to per cpu memory are usually the fastest possible. The per cpu cache lines will stay in the cpu caches for a long time—even if they are dirtied—since no other processor will invalidate the cache lines by writing to per cpu variables of another processor.[6]

A typical use of per cpu data is to manage information about available local memory. If a process requires memory and we can satisfy it from local memory that is tracked via structures in per cpu memory then the performance of the allocator will be optimal. Most of the Linux memory allocators are structured in such a way to minimize access to shared memory locations. Typically it takes a significant imbalance in memory use for an allocator to start assigning memory that is shared with other processors. The sweet point in terms of scalability is encountered when the allocator can keep on serving only local memory.

[4] See Chapter 10, "Spin-locked Kernels," in [12].

[5] For more details on synchronization under Linux, see [5].

[6] Not entirely true. In special situations (for example setup and tear down of per cpu areas) such writes will occur.

Another use of per cpu memory is to keep statistics. Maintaining counters about resource use in the system is necessary for the operating system to be able to adjust to changing computing loads. However, these counters should not impact performance negatively. For that reason Linux keeps essential counters in per processor areas. These counters are periodically consolidated in order to maintain a global state of memory in the system.

The natural use of per cpu data is the maintenance of information about the processor stats and the environment of the processor. This includes interrupt handling, where local memory can be found, timer information as well as other hardware information.

3.4 Per node structures

A node in the NUMA world refers to a section of the system that has its own memory, processors and I/O channels. Per node structures are not as lightweight as per cpu variables because multiple processors on one node may use that per node information. Synchronization is required. However, per node accesses stay within the same hardware enclosure meaning that per node references are to local memory which is more efficient than accessing memory on other nodes. It is advantageous if only local processors use the per node structures. But other remote processors from other nodes may also use any per node structures since we already need locks to provide exclusion for the local processors. Performance is acceptable as long as the use from remote processors is not excessive.

It is natural to use per node structures to manage the resources of such a NUMA node. Allocators typically have first of all per cpu queues where some objects are held ready for immediate access. However, if those per cpu queues are empty then the allocators will fall back to per node resources and attempt to fill up their queues first from the local node and then—if memory gets tight on one node—from remote nodes.

Performance is best if the accesses to per node structures stay within the node itself. Off node allocation scenarios usually involve a degradation in system performance but that may be tolerable given particular needs of an application. Applications that must access more memory than available on one node will have to deal with the effects of intensive off node memory access traffic. In that case it may be advisable to spread out the memory accesses evenly via memory policies in order to not overload a single node.

3.5 Lock locality

In a large system the location of locks is a performance critical element. Lock acquisition typically means gaining exclusive access to a cache line that may be heavily contended. Some processors in the system may be nearer to the cache line than others. These will have an advantage over the others that are more remote. If the cache line becomes heavily contended then processes on remote nodes may not be able to make much progress (starvation). It is therefore imperative that the system implement some way to give each processor a fair chance to acquire the cache line. Frequently such an algorithm is realized in hardware. The hardware solutions have turned out to be effective so far on the platforms that support high processor counts. It is likely though that commodity hardware systems now growing into the space, earlier only occupied by the highly scalable platforms, will not be as well behaved. Recent discussions on the Linux kernel mailing lists indicate that these may not come with the advanced hardware that solve the lock locality issues. Software solutions to this problem—like the hierarchical back off lock developed by Zoran Radovic—may become necessary [10].

3.6 Atomic operations

Atomic operations are the lowest level synchronization primitives. Atomic operations are used as building blocks for higher level constructs. The locks mentioned earlier are examples of such higher level synchronization constructs that are realized using atomic operations.

Linux defines a rich set of atomic operations that can be used to improvise new forms of locking. These operations include both bit operations and atomic manipulation of integers. The atomic operation themselves can be used to synchronize events if they are used to generate state transitions. However, the available state transitions are limited and the set of state transitions observable varies from processor to processor. A library of widely available state transitions via atomic operations has been developed over time. Common atomic operations must be supported by all processors supported by Linux. However, some of the rarer breeds of processors

may not support all necessary atomic operations. Emulation of some atomic operations using locking may be necessary. Ironically the higher level constructs are then used to realize low level atomic operations.

Atomic operations are the lowest level of access to synchronization. Many of the performance critical data structures in Linux are customarily modified using atomic operations that are wrapped using macros. For example the state of the pages in Linux must be modified in such a way. Kernel components may rely on state transitions of these flags for synchronization.

The use of these lower level atomic primitives is complex and therefore the use of atomic operations is typically reserved for performance critical components where enough human resources are available to maintain such custom synchronization schemes. If one of these schemes turns out to be unmaintainable then it is usually replaced by a locking scheme based on higher level constructs.

3.7 Reference counters

Reference counters are a higher level construct realized in Linux using atomic operations. Reference counters use atomic increment and decrement instructions to track the number of uses of an object in the kernel, that way concurrent operation on objects can be performed. If a user of the structure increments the reference counter then the object can be handled with the knowledge that it cannot concurrently be freed. The user of a structure must decrement the reference counter when the object is no longer needed.

The state transition to and from zero is of particular importance here since a zero counter is usually used to indicate that no references exist anymore. If a reference counter reaches zero then an object can be disposed and reclaimed for other uses.

One of the key resources managed using reference counters are the operating system pages themselves. When a page is allocated then it is returned from the page allocator with a reference count of one. Over the lifetime multiple references may be established to the page for a variety of purposes. For example multiple applications may map the same memory page into their process memory. The function to drop a reference on a page checks whether the reference count has reached zero. If so then the page is returned to the page allocator for other uses.

One problem with reference counters is that they require write access to a cache line in the object. Continual establishment of new references and the dropping of old references may cause cache line contention in the same way as locking. Such a situation was recently observed with the zero page on a 1024 processor machine. A threaded application began to read concurrently from unallocated memory (which causes references to the zero page to be established). It took a long time for the application to start due to the cache line with the reference counter starting to bounce back and forth between the caches of various processors that attempted to increment or decrement the counter. Removal of reference counting for the zero page resulted in dramatic improvements in the application startup time.

The establishment of a reference count on an object is usually not sufficient in itself because the reference count only guarantees the continued existence of the object. In order to serialize access to attributes of the object, one still will have to implement a locking scheme. The pages in Linux have an additional page lock that has to be taken in order to modify certain page attributes. The synchronization of page attributes in Linux is complex due to the interaction of the various schemes that are primarily chosen for their performance and due to the fluctuation over time as the locking schemes are modified.

3.8 Read-Copy-Update

RCU is yet another method of synchronization that becomes more and more widespread as the common locking schemes begin to reach their performance limits. The main person developing the RCU functionality for Linux has been Paul McKenney.[7] The main advantage of RCU over a reference counter is that object existence is guaranteed without reference counters. No exclusive cache line has to be acquired for object access which is a significant performance advantage.

RCU accomplishes that feat through a global serialization counter that is used to establish when an object can be freed. The counter only reaches the next state when

[7] See his website at `http://www.rdrop.com/users/paulmck/RCU/`. Retrieved 12 April, 2007. A recent publication is [3], which contains an extensive bibliography.

no references to RCU objects are held by a process. Objects can be reclaimed when they have been expired. All processes referring to the object must have only referenced the object in earlier RCU periods.

RCU is frequently combined with the use of other atomic primitives as well as the exploiting of the atomicity of pointer operations. The combination of atomic operations and RCU can be tricky to manage and it is not easy to develop a scheme that is consistent and has no "holes" where a data structure can become inconsistent. Projects to implement RCU measures for key system components can take a long time. For example the project to develop a lockless page cache using RCU has already taken a couple of years.[8]

3.9 Cache line aliasing / placement

Another element necessary to reach high performance is the careful placement of data into cache lines. Acquiring write access to a cache line can cause a performance issue because it requires exclusive access to the cache line. If multiple unrelated variables are placed in the same cache line then the performance of the access to one variable may be affected by frequents updates of another (false aliasing) because the cache line may need to be frequently reacquired due to eviction to exclusive accesses by other processors. A hotly updated variable may cause a frequently read variable to become costly to access because the cache line cannot be continually kept in the cache hierarchy. Linux solves this issue by providing a facility to arrange variables according to their access patterns. Variables that are commonly read and rarely written to can be placed in a separate section through a special attribute. The cache lines from the mostly read section can then be kept in the caches of multiple processors and are rarely subject to expulsion due to a write request.

Fields of key operating system structures are similarly organized based on common usage and frequency of usage. If two fields are frequently needed in the same function then it is advantageous to put the fields next to each other which increases the chance that both are placed in the same cache line. Access to one field makes the other one available. It is typical to place frequently used data items at the head of a structure to have as many as possible available with a single cache line fetch. In order to guarantee the proper cache line alignment of the fields it is customary to align the structure itself on a cache line boundary.

If one can increase the data density in the cache lines that are at the highest level of the cpu cache stack then performance of the code will increase. Rearranging data in proper cache lines is an important measure to reach that goal.

3.10 Controlling memory allocation

The arrangement in cache lines increases the density of information in the cpu cache and can be used to keep important data near to the processor. In a large system, memory is available at various distances to a processor and the larger the system the smaller the amount of memory with optimal performance for a processor. The operating system must attempt to provide fast memory so that the processes running on the processor can run efficiently.

However, the operating system can only provide heuristics. The usual default is to allocate memory as local to the process as possible. Such an allocation method is only useful if the process will keep on running exclusively on the initial processor. Multithreaded applications may run on multiple processors that may have to access a shared area of memory. Care must be taken about how shared memory is allocated. If a process is started on a particular processor and allocates the memory it needs then the memory will be local to the startup processor. The application may then spawn multiple threads that work on the data structures allocated. These new processes may be moved to distant processors and will now overwhelmingly reference remote memory that is not placed optimally. Moreover all new processes may concurrently access the memory allocated on the node of the initial processor which may exhaust the processing power of the single memory node.

It is advisable that memory be allocated differently in such scenarios. A common solution is to spread the memory out over all nodes that run processes for the application. This will balance the remote cache line processing load over the system. However, the operating system has no way of knowing what the processes of the application will do. Linux has a couple of subsystems that allow the processes to specify memory allocation policies and allocation constraints for a process. Memory can be placed optimally if an application sets up the

[8] See the earlier mentioned work by Nick Piggin, [9].

proper policies depending on how it will access the data. However, this memory control mechanism is not standardized. One will have to link programs to special libraries in order to make use of these facilities. There are new languages on the horizon though that may integrate the locality specification into the way data structures are defined.[9] These new languages may eventually standardize the specification of allocation methods and avoid the use of custom libraries.

3.11 Memory coverage of Translation Lookaside Buffers (TLB)

Each of the processes running on modern processors has a virtual address space context. The address space context is provided by TLB entries that are cached by the processor in order to allow a user process access to physical memory. The amount of TLB entries in a processor is limited and the limit on the number of TLB entries in turn limits the amount of physical memory that a processor may access without incurring a TLB miss. The size of available physical memory is ever growing and so the fraction of memory physically accessible without a TLB miss is ever shrinking.

Under Linux, TLB misses are a particular problem since most architectures use a quite small page size of 4 kilobytes. The larger systems support 16 kilobytes. On the smaller systems—even with a thousand TLB entries—one will only be able to access 4 megabytes without a TLB miss. TLB miss overhead varies between processors and ranges from a few dozen clock cycles if the corresponding page table entry is in the cache (Intel-64) to hundreds and occasionally even a few thousand cycles on machines that require the implementation of TLB lookups as an exception handler (like IA64).

For user processes, Linux is currently restricted to a small 4k page size. In kernel space an attempt is made to directly map all of memory via 1-1 mappings. These are TLB entries that provide no translation at all. The main use of these TLBs is to specify the access parameters for kernel memory. Many processors also support a larger page size. It is therefore common that the kernel itself use larger TLB entries for its own memory. This increases the TLB coverage when running in kernel mode significantly. The sizes in use on larger Linux machines (IA64) are 16M TLB entries whereas the smaller (Intel-64 based) machines provide 2M TLB entries to map kernel memory.

In order to increase the memory coverage, another subsystem has been added to Linux that is called the *hugetlb file system*. On Intel-64 this will allow the management of memory mapped via 2M TLB entries. On IA64 memory can be managed in a variety of sizes from 2M to 1 Gigabytes. However, hugetlb memory cannot be treated like regular memory. Most importantly files cannot be memory mapped using hugetlbfs. I/O is only possible in 4 kilobyte blocks through buffered file I/O and direct I/O. Projects are underway to use huge pages for executables and provide transparent use of huge pages for process data [6].

A microkernel would require the management of additional address spaces via additional TLB entries that would compete for the limited TLB slots in a processor. TLB pressure would increase and we would have more overhead coming about through the separate address spaces of a microkernel that would degrade performance.

4 Multicore / Chip Multithreading

Recent developments are leading to increased multi threading on a single processor. Multiple cores are placed on a single chip. The inability to increase the clock frequency of processors further leads to the development of processors that are able to execute a large number of threads concurrently. In essence we see the miniaturization of contemporary supercomputers on a chip. The complex interaction of the memory caches of multi core processors will present additional challenges to organizing memory and to balancing of a computing load to run with maximum efficiency. It seems that the future is owned by multithreaded applications and operating system kernels that have to use complex synchronization protocols in order to extract the maximum performance from the available computational resources.

Rigid microkernel concepts require isolation of kernel subsystems. It is likely going to be a challenge to implement complex locking protocols between kernel components that can only communicate via messages or some form of inter process communication. Instead processes wanting to utilize the parallel execution capabilities to the fullest must have a shared address space in which it is possible to realize locking schemes as needed to deal with the synchronization of the individual tasks.

[9] As realized for example in Unified Parallel C.

5 Process Contention for System Resources

The scaling of individual jobs on a large system depends on the use of shared resources. Processes that only access local resources and that have separate address spaces run with comparable performance to that on smaller machines since there is minimal locking overhead. On a machine with a couple of thousand processors, one can run a couple of thousand independent processes that all work with their own memory without scaling concerns. This ability shows that the operating system itself has been optimized to fully take advantage of process isolation for scaling. The situation becomes different if all these processes share a single address space. In that case certain functions—like the mapping of a page into the common memory space of these processes—must be serialized by the operating system. Performance bottlenecks can result if many of the processes perform operations that require the same operating system resource. At that point the synchronization mechanisms of the operating system become key to reduce the performance impact of contention for operating system resources.

However, the operating system itself cannot foresee, in detail, how processes will behave. Policies can be specified describing how the operating system needs to manage resources but the operating system itself can only provide heuristics for common process behavior. Invariably sharing resources in a large supercomputer for complex applications requires careful planning and proper setup of allocation policies so that bottleneck can be avoided. It is necessary to plan how to distribute shared memory depending on the expected access patterns to memory and common use of operating system resources. Applications can be run on supercomputers without such optimizations but then memory use, operating system resource use may not be optimal.

6 Conclusion

A monolithic operating system such as Linux has no restrictions on how locking schemes can be developed. A unified address space exists that can be accessed by all kernel components. It is therefore possible to develop a rich multitude of synchronization methods in order to make best use of the processor resources. The freedom to do so has been widely used in the Linux operating system to scale to high processor counts. The locking methodology can be varied and may be alternatively coarse grained or more refined depending on the performance requirements for a kernel component. Critical operating system paths can be successively refined or even be configurable for different usage scenarios. For example the page table locking scheme in Linux is configurable depending on the number of processors. For a small number of processors, there will be only limited contention on page table and therefore a single page table lock is sufficient. If a large number of processors exists in a system then contention may be an issue and having smaller grained locks is advantageous. For higher processor counts the Linux kernel can implement a two tier locking scheme where the higher page table layers are locked by a single lock whereas the lowest layer has locks per page of page table entries. The locking scheme becomes more complicated—which will have a slight negative performance impact on smaller machines—but provides performance advantages for highly concurrent applications.

As a result, the Linux operating system as a monolithic operating system can adapt surprisingly well to high processor counts and large memory sizes. Performance bottlenecks that were discovered while the system was gradually scaled up to higher and higher processor counts were addressed through alternating approaches using a variety of locking approaches. In 2007 Linux supports up to 4096 processors with around 16 terabytes of memory on 1024 nodes. Configurations of up to 1024 processors are supported by commercial Linux distributions. There are a number of supercomputer installation that use these large machines for scientific work at the boundaries of contemporary science.

The richness of the locking protocols that made the scaling possible requires an open access policy within the kernel. It seems that microkernel based designs are fundamentally inferior performance-wise because the strong isolation of the components in other process contexts limits the synchronization methods that can be employed and causes overhead that the monolithic kernel does not have to deal with. In a microkernel data structures have to be particular to a certain subsystem. In Linux data structures may contain data from many subsystems that may be protected by a single lock. Flexibility in the choice of synchronization mechanism is core to Linux success in scaling from embedded systems to supercomputers. Linux would never have been able to scale to these extremes with a microkernel based approach because of the rigid constraints that strict mi-

crokernel designs place on the architecture of operating system structures and locking algorithms.

References

[1] Catanzaro, Ben. *Multiprocessor Systems Architectures: A Technical Survey of Multiprocessor/ Multithreaded Systems using SPARC, Multilevel Bus Architectures and Solaris (SunOS).* Mountain View: Sun Microsystems, 1994.

[2] El-Ghazawi, Tarek, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming.* Wiley Interscience, 2003.

[3] Hart, Thomas E., Paul E. McKenney, and Angela D. Brown. *Making Lockless Synchronization Fast: Performance Implications of Memory Reclaim.* Parallel and Distributed Processing Symposium, 2006.

[4] Hwang, Kai and Faye A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York: 1984.

[5] Lameter, Christoph. *Effective Synchronization on Linux/NUMA Systems.* Palo Alto: Gelato Foundation, 2005. Retrieved April 11, 2006. `http://kernel.org/pub/linux/kernel/people/christoph/gelato/gelato2005-paper.pdf`

[6] H.J. Lu, Rohit Seth, Kshitij Doshi, and Jantz Tran. "Using Hugetlbfs for Mapping Application Text Regions" in *Proceedings of the Linux Symposium: Volume 2*. pp. 75–82. (Ottawa, Ontario: 2006).

[7] Milojic, Dejan S. *Implementation for the Mach Microkernel.* Friedrich Vieweg & Sohn Verlag, 1994.

[8] Mosberger, David. Stephane Eranian. *ia-64 linux kernel: design and implementation.* New Jersey: Prentice Hall, 2002.

[9] Piggin, Nick. "A LockLess Page Cache in Linux" in *Proceedings of the Linux Symposium: Volume 2* (Ottawa, Ontario: 2006). Retrieved 11 April 2006. `http://ols2006.108.redhat.com/reprints/piggin-reprint.pdf`.

[10] Radovic, Zoran. *Software Techniques for Distributed Shared Memory.* Uppsala: Uppsala University, 2005, pp. 33–54.

[11] Roch, Benjamin. *Monolithic kernel vs. Microkernel.* Retrieved 9 April 2007. `http://www.vmars.tuwien.ac.at/courses/akti12/journal/04ss/article_04ss_Roch.pdf`.

[12] Schimmel, Kurt. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers.* New York: Addison-Wesley, 1994.

[13] Tannenbaum, Andrew S. *Modern Operating Systems.* New Jersey: Prentice Hall, 1992.

[14] Tannenbaum, Andrew S., Albert S. Woodhul. *Operating Systems Designs and Implementation* (3rd Edition). New Jersey: Prentice-Hall, 2006.

Performance and Availability Characterization for Linux Servers

Vasily Linkov
Motorola Software Group
Vasily.Linkov@motorola.com

Oleg Koryakovskiy
Motorola Software Group
Oleg.Koryakovskiy@motorola.com

Abstract

The performance of Linux servers running in mission-critical environments such as telecommunication networks is a critical attribute. Its importance is growing due to incorporated high availability approaches, especially for servers requiring five and six nines availability. With the growing number of requirements that Linux servers must meet in areas of performance, security, reliability, and serviceability, it is becoming a difficult task to optimize all the architecture layers and parameters to meet the user needs.

Other Linux servers, those not operating in a mission-critical environment, also require different approaches to optimization to meet specific constraints of their operating environment, such as traffic type and intensity, types of calculations, memory, and CPU and IO use.

This paper proposes and discusses the design and implementation of a tool called the Performance and Availability Characterization tool, PAC for short, which operates with over 150 system parameters to optimize over 50 performance characteristics. The paper discusses the PAC tool's architecture, multi-parametric analysis algorithms, and application areas. Furthermore, the paper presents possible future work to improve the tool and extend it to cover additional system parameters and characteristics.

1 Introduction

The telecommunications market is one of the fastest growing industries where performance and availability demands are critical due to the nature of real-time communications tasks with requirement of serving thousands of subscribers simultaneously with defined quality of service. Before Y2000, telecommunications infrastructure providers were solving performance and availability problems by providing proprietary hardware and software solutions that were very expensive and in many cases posed a lock-in with specific vendors. In the current business environment, many players have come to the market with variety of cost-effective telecommunication technologies including packed data technologies such as VoIP, creating server-competitive conditions for traditional providers of wireless types of voice communications. To be effective in this new business environment, the vendors and carriers are looking for ways to decrease development and maintenance costs, and decrease time to market for their solutions.

Since 2000, we have witnessed the creation of several industry bodies and forums such as the Service Availability Forum, Communications Platforms Trade Association, Linux Foundation Carrier Grade Linux Initiative, PCI Industrial Computer Manufacturers Group, SCOPE Alliance, and many others. Those industry forums are working on defining common approaches and standards that are intended to address fundamental problems and make available a modular approach for telecommunication solutions, where systems are built using well defined hardware specifications, standards, and Open Source APIs and libraries for their middleware and applications [11] ("Technology Trends" and "The .org player" chapters).

The Linux operating system has become the *de facto* standard operating system for the majority of telecommunication systems. The Carrier Grade Linux initiative at the Linux Foundation addresses telecommunication system requirements, which include availability and performance [16].

Furthermore, companies as Alcatel, Cisco, Ericsson, Hewlett-Packard, IBM, Intel, Motorola, and Nokia use Linux solutions from MontaVista, Red Hat, SuSE, and WindRiver for such their products as softswitches, telecom management systems, packet data gateways, and routers [13]. Examples of existing products include Alcatel Evolium BSC 9130 on the base of the Advanced TCA platform, and Nortel MSC Server [20], and many

more of them are announced by such companies as Motorola, Nokia, Siemens, and Ericsson.

As we can see, there are many examples of Linux-based, carrier-grade platforms used for a variety of telecommunication server nodes. Depending on the place and functionality of the particular server node in the telecommunication network infrastructure, there can be different types of loads and different types of performance bottlenecks.

Many articles and other materials are devoted to questions like "how will Linux-based systems handle performance critical tasks?" In spite of the availability of carrier-class solutions, the question is still important for systems serving a large amount of simultaneous requests, e.g. WEB Servers [10] as Telecommunication-specific systems.

Telecommunication systems such as wireless/mobile networks have complicated infrastructures implemented, where each particular subsystem solves its specific problem. Depending on the problem, the critical systems' resource could be different. For example, Dynamic Memory Allocation could become a bottleneck for Billing Gateway, Fraud Control Center (FCC), and Data Monitoring (DMO) [12] even in SMP architecture environment. Another example is WLAN-to-WLAN handover in UMTS networks where TCP connection re-establishment involves multiple boxes including HLR, DHCP servers and Gateways, and takes significant time (10–20 sec.) which is absolutely unacceptable for VoIP applications [15]. A similar story occurred with TCP over CDMA2000 Networks, where a bottleneck was found in the buffer and queue sizes of a BSC box [17]. The list of the examples can be endless.

If we consider how the above examples differ, we would find out that in most cases performance issues appear to be quite difficult to deal with, and usually require rework and redesign of the whole system, which may obviously be very expensive.

The performance improvement by itself is quite a well-known task that is being solved by the different approaches including the Clustering and the Distributed Dynamic Load Balancing (DDLB) methods [19]; this can take into account load of each particular node (CPU) and links throughput. However, a new question may arise: "Well. We know the load will be even and dynamically re-distributed, but what is the maximum system performance we can expect?" Here we are talking not about performance problems, but about performance characterization of the system. In many cases, people working on the new system development and fortunately having performance requirements agreed up front use prototyping techniques. That is a straightforward but still difficult way, especially for telecommunication systems where the load varies by types, geographic location, time of the day, etc. Prototyping requires creation of an adequate but inexpensive model which is problematic in described conditions.

The authors of this paper are working in telecommunication software development area and hence tend to mostly consider problems that they face and solve in their day-to-day work. It was already said that performance issues and characterization are within the area of interest for a Linux-based system developer. Characterization of performance is about inexpensive modeling of the specific solution with the purpose of predicting future system performance.

What are the other performance-related questions that may be interesting when working in telecommunications? It isn't just by chance we placed the word *Performance* close to *Availability*; both are essential characteristics of a modern telecommunication system. If we think for a moment about the methods of achieving of some standard level of availability (let's say the five- or six- nines that are currently common industry standards), we will see that it is all about redundancy, reservation, and recovery. Besides specific requirements to the hardware, those methods require significant software overhead functionality. That means that in addition to system primary functions, it should provide algorithms for monitoring failure events and providing appropriate recovery actions. These algorithms are obviously resource-consuming and therefore impact overall system performance, so another problem to consider is a reasonable tradeoff between availability and productivity [8], [18].

Let's consider some more problems related to telecommunication systems performance and availability characterization that are not as fundamental as those described above, but which are still important (Figure 1).

Performance profiling. The goal of performance profiling is to verify that performance requirements have been achieved. Response times, throughput, and other time-sensitive system characteristics should be measured and evaluated. The performance profiling is ap-

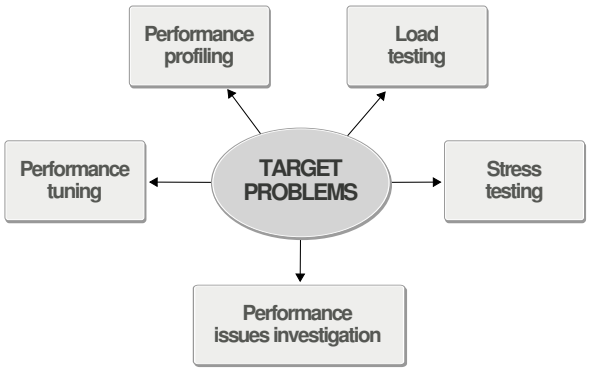


Figure 1: Performance and Availability Characterization target problems

plicable for release-to-release testing.

Load testing. The goal of load testing is to determine and ensure that the system functions properly beyond the expected maximum workload. The load testing subjects the system to varying workloads to evaluate the performance behaviors and ability of the system to function properly under these different workloads. The load testing also could be applicable on design stage of a project to choose the best system architecture and ensure that requirements will be achieved under real/similar system workloads [21], [22].

Stress testing. The goal of stress testing is to find performance issues and errors due to low resources or competition for resources. Stress testing can also be used to identify the peak workload that the system can handle.

Performance issue investigation. Any type of performance testing in common with serious result analysis could be applicable here. Also in some cases, snapshot gathering of system characteristics and/or profiling could be very useful.

Performance Tuning. The goal of performance tuning is to find optimal OS and Platform/Application settings, process affinity, and schedule policy for load balancing with the target of having the best compromise between performance and availability. The multi-objective optimization algorithm can greatly reduce the quantity of tested input parameter combinations.

This long introduction was intended to explain why we started to look at performance and availability characterization problems and their applications to Linux-based, carrier-grade servers. Further along in this paper, we will consider existing approaches and tools and share one more approach that was successfully used by the authors in their work.

2 Overview of the existing methods for Performance and Availability Characterization

A number of tools and different approaches for performance characterization exist and are available for Linux systems. These tools and approaches target different problems and use different techniques for extracting system data to be analyzed as well, and support different ways to represent the results of the analysis. For the simplest cases of investigating performance issues, the standard Linux tools can be used by anyone. For example, the GNU profiler ***gprof*** provides basic information about pieces of code that are consuming more time to be executed, and which subprograms are being run more frequently than others. Such information offers understanding where small improvements and enhancements can give significant benefits in performance. The corresponding tool ***kprof*** gives an opportunity to analyze graphical representation gprof outputs in form of call-trees, e.g. comprehensive information about the system can be received from ***/proc*** (a reflection of the system in memory). Furthermore, for dealing with performance issues, a variety of standard debugging tools such as instrumentation profilers (***oprofile*** which is a system-wide profiler), debuggers ***kdb*** and ***kgdb***, allowing kernel debugging up to source code level as well as probes crash dumps and many others are described in details in popular Linux books [3]. These tools are available and provide a lot of information. At the same time a lot of work is required to filter out useful information and to analyze it. The next reasonable step that many people working on performance measurement and tuning attempt to do is to create an integrated and preferably automated solution which incorporates in it the best features of the available standalone tools.

Such tools set of benchmarks and frameworks have appeared such as the well known package ***lmbench***, which is actually a set of utilities for measurement of such characteristics as memory bandwidth, context switching, file system, process creating, signal handling latency, etc. It was initially proposed and used as a universal performance benchmarking tool for Unix-based systems. There were several projects intended to develop new microbenchmark tools on the basis of ***lmbench*** in order to improve measurement precision and applicability for low-latency events by using high-resolution

timers and internal loops with measurement of the average length of events calculated through a period of time, such as ***Hbench-OS*** package [4]. It is noticeable that besides widely used performance benchmarks, there are examples of availability benchmarks that are specifically intended to evaluate a system from the high availability and maintainability point of view by simulating failure situations over a certain amount of time and gathering corresponding metrics [5].

Frameworks to run and analyze the benchmarks were the next logical step to customize this time-consuming process of performance characterization. Usually a framework is an automated tool providing additional customization, automation, representation, and analysis means on top of one or several sets of benchmarks. It makes process of benchmarking easier, including automated decision making about the appropriate amount of cycles needed to get trustworthy results [23].

Therefore, we can see that there are a number of tools and approaches one may want to consider and use to characterize a Linux-based system in terms of performance. Making the choice we always keep in mind the main purpose of the performance characterization. Usually people pursue getting these characteristics in order to prove or reject the assumption that a particular system will be able to handle some specific load. So if you are working on a prototype of a Linux-based server for use as a wireless base site controller that should handle e.g. one thousand voice and two thousand data calls, would you be happy to know from the benchmarks that your system is able to handle e.g. fifty thousand TCP connections? The answer isn't trivial in this case. To make sure, we have to prepare a highly realistic simulated environment and run the test with the required number of voice and data calls. It is not easy, even if the system is already implemented, because you will have to create or simulate an external environment that is able to provide an adequate type and amount of load, and which behaves similarly to a live wireless infrastructure environment. In case you are in the design phase of your system, it is just impossible. You will need to build your conclusion on the basis of a simplified system model. Fortunately, there is another approach—to model the load, not the system. Looking at the architecture, we can assume what a specific number of voice and data calls will entail in the system in terms of TCP connections, memory, timers, and other resources required. Having this kind of information, we can use benchmarks for the identified resources and make the conclusion after running and analyzing these benchmarks on the target HW/SW platform, without the necessity of implementing the application and/or environment. This approach is called workload characterization [2].

Looking back to the Introduction section, we see that all the target questions of Performance and Availability characterization are covered by the tools we have briefly looked through above. At the same time there is no single universal tool that is able to address all these questions. Further in the paper we are introducing the Performance and Availability Characterization (PAC) tool that combines the essential advantages of all the approaches considered in this chapter and provides a convenient framework to perform comprehensive Linux-based platforms characterization for multiple purposes.

3 Architectural Approach

3.1 Experimental Approach

Anyone who is trying to learn about the configuration of Linux servers running in mission-critical environments and running complex applications systems will have to address the following challenges:

- An optimal configuration, suitable for any state of environmental workload, does not exist;
- Systems are sophisticated: Distributed, Multiprocessor, Multithreaded;
- Hundreds or even thousands of configuration parameters can be changed;
- Parameters can be poorly documented, so the result of a change for a group of parameters or even single parameter can be totally unpredictable.

Based on the above described conditions, an analytical approach is scarcely applicable, because a system model is not clear. An empirical approach could be more applicable to find optimal configuration of a system, but only experimental evaluation can be used to validate the correctness of optimal configuration on a real system. The heart of PAC is the concept of the experimentation. A single experiment consists of the following parts:

- Input parameters: let us call them Xs. Input parameters are all that you want to set up on a target system. Typical examples here are Linux kernel variables, loader settings, and any system or application settings.
- Output parameters: let us call them Ys. Output parameters are all that you want to measure or gather on a target system: CPU and Memory utilization, any message throughput and latency, system services bandwidth, and more. Sources for Ys could be: `/proc` file system, loaders output, profiling data, and any other system and application output.
- Experiment scenarios: An experiment scenario is a description of actions which should be executed on target hosts.

Typical experiment scenario follows a sequence of action: setup Xs that can't be applied on-the-fly (including execution required actions to apply such Xs like restart node or processes, down and up network interfaces, etc.), then setup Xs that can be applied on-the-fly and loader's Xs, start loaders, next setup Xs like: schedule policy, priority, CPU binding etc., finally collect Ys such as CPU/Memory usage, stop loaders, and overall statistics.

Every scenario file may use preprocessor directives and ***S-Language*** statements. ***S-Language*** is a script language which is introduced specifically for the project. Both preprocessor and ***S-Language*** are described in more detail following. One of the important parts of the scenario executor is a dynamic table of variables. Variable is a pair-variable name and variable value. There are two sources of the variables in the dynamic table:

- Xs (Input variables). They are coming from an experiment.
- Ys (Collected variables). They are coming from remote hosts.

In the case of input variables, the names of the variables are provided by the XML-formatted single experiment file. In the case of collected variables, the names of the variables are provided by scripts or other executables on the target hosts' side. Whenever the same executable could be run on many different hosts, a namespace mechanism is introduced for the variable names. A host identifier is used as a namespace of the variable name.

3.2 Overview of PAC Architecture

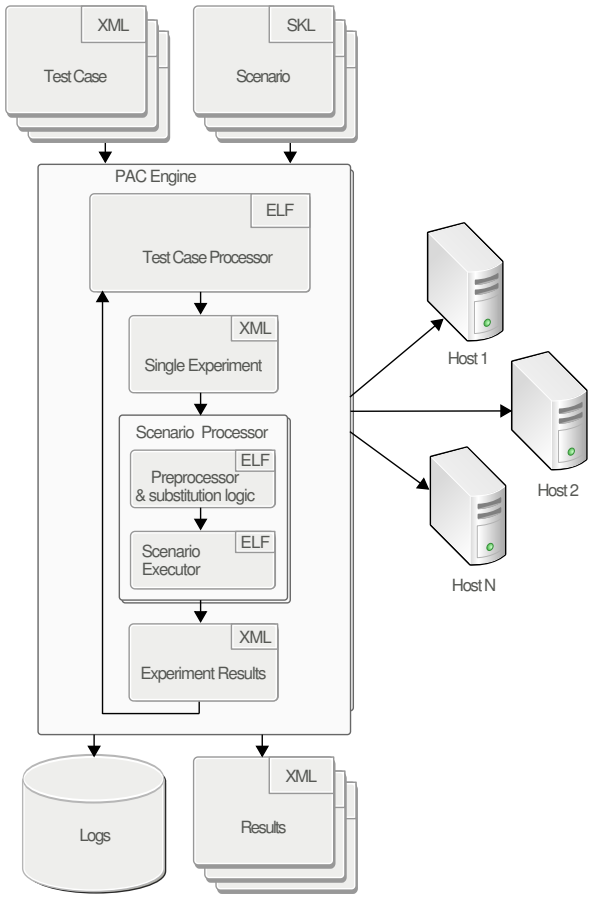


Figure 2: PAC Architecture

A test case consists of a set of experiments. Each experiment is essentially a set of Xs that should be used while executing a scenario. Set of Xs within one Test Case boundaries is constant and only values of these Xs are variable. Each experiment is unambiguously linked to a scenario. A scenario resides in a separate file or in a group of files. The PAC engine overall logic is as follows:

- Takes a test case file;
- For each experiment in the test case, performs the steps below;
- Selects the corresponding scenario file;
- Executes all the scenario instructions using settings for fine tuning the execution logic;
- Saves the results into a separate result file;

- Saves log files where the execution details are stored.

Test Cases The basic unit of test execution is an experiment. A single experiment holds a list of variables, and each variable has a unique name. Many experiments form a test case. The purpose of varying Xs' values depends on a testing goal. Those Xs' names are used in scenarios to be substituted with the values for a certain experiment.

Scenarios A scenario is a description of actions which should be executed on target hosts in a sequence or in parallel in order to set up Xs' values in accordance with Test Case/Experiments and gather the values of Ys. The solution introduces a special language for writing scenarios. The language simplifies description of actions that should be executed in parallel on many hosts, data collection, variable values, substitution, etc.

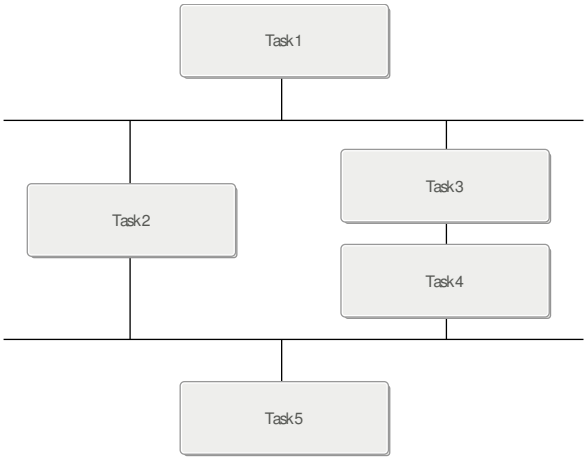
Results The results files are similar to Test Case files. However, they contain set of Ys coming from target hosts and from input experiment variables (Xs).

The scenario processor consists of two stages, as depicted in the figure above. At the bottom line there is a scenario executor which deals with a single scenario file. From the scenario executor's point of view, a scenario is a single file; however, it is a nice feature to be able to group scenario fragments into separate files. To support this feature the preprocessor and substitution logic is introduced in the first stage. The standard C programming language preprocessor is used at this stage, so anything which is supported by the preprocessor can be used in a scenario file. Here is a brief description of the C preprocessor features which is not a complete one and is given here for reference purposes only:

- Files inclusion;
- Macro substitutions;
- Conditional logic.

Summarizing, the complete sequence of actions is as follows: The single experiment from the Test Case is applied to the scenario file. It assumes macro substitutions of the experiment values (Xs), file inclusions, etc. The scenario executor follows instructions from the scenario file. While executing the scenario some variables (Ys) are collected from target hosts. At the end of the scenario execution, two files are generated: a log file and a results file. The log file contains the report on what was executed and when, on which host, as well as the return codes of the commands. The results file contains a set of collected variables (Xs and Ys).

The main purpose of the introduced ***S-Language*** is to simplify a textual description of the action sequences which are being executed consecutively and/or in parallel on many hosts. Figure 3 shows an example task execution sequence.



(a) Block diagram

```
TITLE ``Scenario example''

#include "TestHosts.incl"
#include "CommonDef.incl"

/* Task1 */

WAIT ssh://USER:PASSWD @ {X_TestHost} "set_kernel_tun.sh SEM \
                {X_IPC_SEM_KernelSemmni} {X_IPC_SEM_KernelSemopm}"

PARALLEL
{
       /* Task2 */
       COLLECT @ X_TestHost "sem_loader -d {X_Duration} \
       -r {X_IPC_SEM_NumPVOps} -t {X_IPC_SEM_LoaderNumThreads}"
SERIAL
{
      /* Task3&4 */
      COLLECT @ {X_TestHost} "get_overall_CPUusage.pl -d
{X_Duration}"
      COLLECT [exist(Y_Memory_Free)] @ {X_TestHost} \
      "get_overall_Memoryusage.pl -d X_Duration"
      }
}
/* Task5 */
NOWAIT [IPC_iteration >1] @ {X_TestHost} ``cleanup_timestamps.sh''
```

(b) ***S-Language*** Code

Figure 3: Scenario Example

ExecCommand is a basic statement of the ***S-Language***. It instructs the scenario executor to execute a command on a target host. The non-mandatory ***Condition*** element specifies the condition on when the command is to be executed. There are five supported command modifiers: ***RAWCOLLECT***, ***COLLECT***, ***WAIT***, ***NOWAIT***, and ***IGNORE***. The At Clause part specifies on which

host the command should be executed. The At Clause is followed by a string literal, which is the command to be executed. Substitutions are allowed in the string literal.

3.3 PAC Agents

We are going to refer to all software objects located on a target system as *PAC agents*. The server side of PAC does not contain any Performance and Availability specifics, but it is just intended to support any type of complex testing and test environment. Everybody can use PAC itself to implement their own scenario and target agents in order to solve their own specific problem related to the system testing and monitoring.

PAC agents, which are parts of the PAC tool, are the following:

- Linux service loaders;
- Xs adjusting scripts;
- Ys gathering scripts.

In this paper, we consider only loaders as more interesting part of PAC agents. The diagram in Figure 4 is intended to show the common principle of the loader implementation. Every loader receives a command line

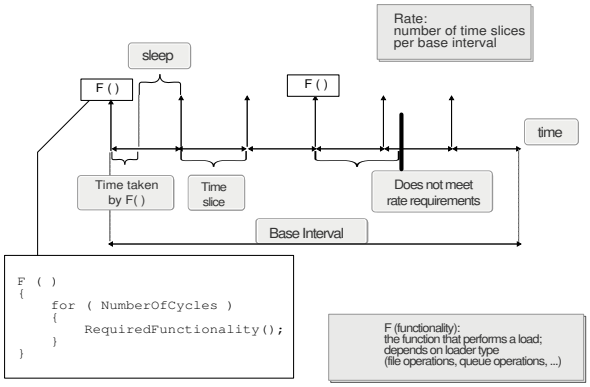


Figure 4: Loader Implementation

argument which provides the number of time slices a base interval (usually one second) is going to be divided into. For example: `<loader> --rate 20` means that a one-second interval will be divided into 20 slices.

At the very beginning of each time slice, a loader calls a function which performs a required functionality/load. The functionality depends on a loader type. For example, the file system loader performs a set of file operations, while the shared memory loader performs a set of shared memory operations, and so on. If the required functionality has been executed before the end of the given time slice, a loader just sleeps until the end of the slice. If the functionality takes longer than a time slice, the loader increments the corresponding statistic's counter and proceeds.

There are several common parameters for loaders.

Input:

- The first one is a number of threads/processes. The main thread of each loader's responsibility is to create the specified number of threads/processes and wait until they are finished. Each created thread performs the loader-specific operations with the specified rate.
- The second common thing is the total loader working time. This specifies when a loader should stop performing operations.
- Loaders support a parameter which provides the number of operations per one "call of F() functionality." For example, a signal loader takes an argument of how many signals should be sent per time slice. This parameter, together with the number of threads, rate, and number of objects to work with (like number of message queues), gives the actual load.
- Besides that, each loader accepts specific parameters (shared memory block size in kilobytes, message size, and signal number to send, and so on).

Output:

- Number of *fail*s due to the rate requirement not being met.
- Statistics—figures which are specific for a loader (messages successfully sent, operations successfully performed, etc.)

The loaders implementation described above allows not only the identification of Linux service breakpoints, but also—with help of fine rate/load control—the discovery of the service behavior at different system loads and settings. The following loaders are available as part of PAC tool:

- IPC loaders:
  - Shared memory loader;
  - Semaphore loader;
  - Message queues loader;
  - Timer loader.
- CPU loaders:
  - CPU loader;
  - Signal loader;
  - Process loader.
- IP loaders:
  - TCP loader;
  - UDP loader.
- FS loader (File & Storage);
- Memory loader.

One more PAC agent is PPA (Precise Process Accounting). PPA is a kernel patch that has been contributed by Motorola. PPA enhances the Linux kernel to accurately measure user/system/interrupt time both per-task and system wide (all stats per CPU). It measures time by explicitly time-stamping in the kernel and gathers vital system stats such as system calls, context switches, scheduling latency, and additional ones. More information on PPA is available from the PPA SourceForge web site: `http://sourceforge.net/projects/ppacc/`.

4 Performance and Availability Characterization in Use

Let us assume that you already have the PAC tool and you have decided to use it for your particular task. First of all, you will have to prepare and plan your experiments:

- Identify list of input parameters (Xs) that you would like to set up on the target. That could be kernel parameters, a loader setting like operational rate, number of processes/threads, CPU binding, etc.
- Identify list of output parameters (Ys) that you would like to measure during an experiment: everything you want to learn about the system when it is under a given load.

If we are talking about Linux systems, you are lucky then, because you can find in the PAC toolset all the necessary components for the PAC agent that have been already implemented: set scripts for Xs, get scripts for Ys, and predefined scenarios for Linux's every service.

If you are not using Linux, you can easily implement your own scripts, scenarios, and loaders. When you have identified all the parameters that you want to set up and measure, you can move on to plan the experiments to run.

We will start with some semaphore testing for kernels 2.4 and 2.6 on specific hardware. Let's consider the first test case (see Table 1). Every single line represents a single experiment that is a set of input values. You can see some variation for number of semaphores, operation rate, and number of threads for the loader; all those values are Xs. A test case also has names of the values that should be collected—Ys.

As soon as the numbers are collected, let's proceed to the data analysis. Having received results of the experiments for different scenarios, we are able to build charts and visually compare them. Figure 5 shows an example of semaphore charts for two kernels: 2.4 on the top, and 2.6 on the bottom. The charts show that kernel 2.4 has a lack of performance in the case of many semaphores. It is not easy to notice the difference between the kernels without having a similar tool for collecting performance characteristics. The charts in Figure 6 are built from the same data as the previous charts; however, a CPU measurement parameter was chosen for the vertical axis. The charts show that the CPU consumption is considerably less on 2.6 kernel in comparison with 2.4. In the Introduction section, we presented some challenges related to performance and availability. In this section, we will cover how we face these challenges by experiment planning and appropriate data analysis approach.

Performance Profiling

- Set up predefined/standard configuration for the kernel and system services.
- Setup loaders to generate the workload as stated in your requirements.

TC_ID	Experiment_ID	X_IPC_SEM_KernelSemmni	X_IPC_SEM_KernelSemmns	X_IPC_SEM_KernelSemopm	X_IPC_SEM_LoaderNumSems	X_IPC_SEM_NumPVOps	X_IPC_SEM_LoaderNumThreads	Y_CPU_Percentage_cpu1	Y_CPU_Percentage_cpu2	Y_CPU_Percentage_cpu3	Y_CPU_Percentage_cpu4	Y_MEMORY_Free	Y_IPC_SEM_NumSems	Y_IPC_SEM_LdrLockOp	Y_IPC_SEM_LdrUnlockOp	Y_IPC_SEM_LdrRateFailed
30	18	32000	800	1024	300	170	8	1	3	0	0	8061550592	300	52132.7	52132.7	0.0666667
30	19	32000	800	1024	300	300	8	3	0	1	0	8055160832	300	89916.8	89916.8	0
30	20	32000	800	1024	300	800	8	0	3	9	3	8053686272	300	233750	233750	0
30	21	32000	800	1024	300	1700	8	7	0	6	0	8054013952	300	494471	494471	0
30	22	32000	800	1024	300	2000	8	7	14	7	14	8059387904	300	584269	584269	0.0777778
30	23	32000	800	1024	300	2300	8	16	0	8	8	8058470400	300	674156	674156	0.0333333
30	24	32000	800	1024	300	2700	8	19	10	0	9	8062369792	300	791193	791193	0.0666667
30	25	32000	800	1024	1	10000	20	1	0	0	0	8045821952	1	9712.8	9712.8	0
30	26	32000	800	1024	1	50000	20	0	0	0	4	8059224064	1	48474.1	48474.1	0
30	27	32000	800	1024	1	100000	20	8	0	0	0	8068726784	1	96912.2	96912.2	0
30	28	32000	800	1024	1	250000	20	0	21	0	0	8060928000	1	242340	242340	0
30	29	32000	800	1024	1	500000	20	0	41	0	0	8052441088	1	484651	484651	0
30	30	32000	800	1024	1	600000	20	2	47	0	0	8070725632	1	581599	581599	0
30	31	32000	800	1024	1	700000	20	0	57	0	0	8054112256	1	678266	678266	0
30	32	32000	800	1024	1	800000	20	59	0	0	0	8082391040	0	775435	775435	0
30	33	32000	800	1024	100	100	20	0	0	0	0	8063451136	100	9991.11	9991.11	0
30	34	32000	800	1024	100	500	20	1	1	0	0	8058863616	100	50958.4	50958.4	0
30	35	32000	800	1024	100	1000	20	0	0	1	0	8046870528	100	98917.5	98917.5	0
30	36	32000	800	1024	100	2500	20	4	1	3	4	8058142720	100	242667	242667	0
30	37	32000	800	1024	100	5000	20	11	0	0	3	8047525888	100	485289	485289	0
30	38	32000	800	1024	100	6000	20	8	9	0	9	8052932608	100	584133	584133	0

Table 1: Table of Experiments

- Perform experiments.
- Check whether the measured data shows that requirements are met.

Load Testing

- Set up predefined/standard configuration for the kernel and system services.
- Use a long experiment duration.
- Mix the workload for all available services.
- Vary workloads.
- Vary the number of threads and instances for the platform component.
- Analyze system behavior.
- Check that Ys are in valid boundaries.

Stress Testing

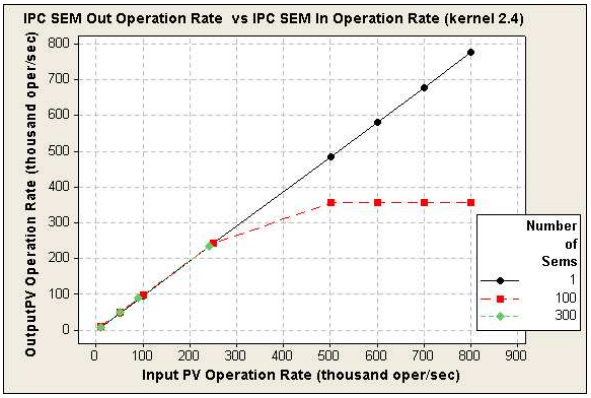
- Use a high workload.
- Operate by the loader with the target to exhaust system resources like CPU, memory, disk space, etc.
- Analyze system behavior.
- Check that all experiments are done and Ys are in valid boundaries.

Performance tuning

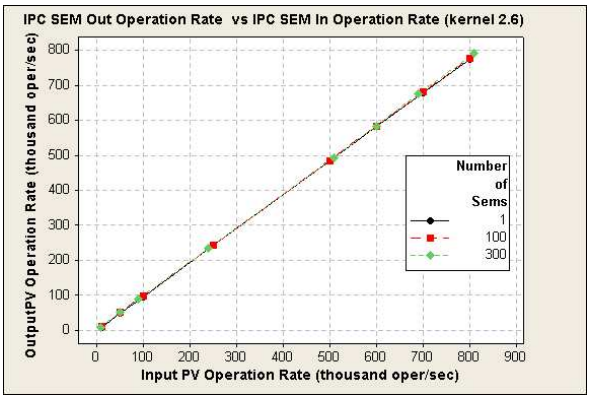
- Plan your experiments from a workload perspective with the target to simulate a real load on the system.
- Vary Linux settings, process affinity, schedule police, number of threads/instances, etc.
- Analyze the results in order to identify the optimal configuration for your system. Actually we believe a multi-objective optimization can be very useful for that. This approach is described in more detail later on.

System Modeling

- Take a look at your design. Count all the system objects that will be required from an OS perspective, like the number of queues, TCP/UDP link, timers, semaphores, shared memory segment, files, etc.
- Examine your requirements in order to extrapolate this on every OS service workload.
- Prepare test case(s).

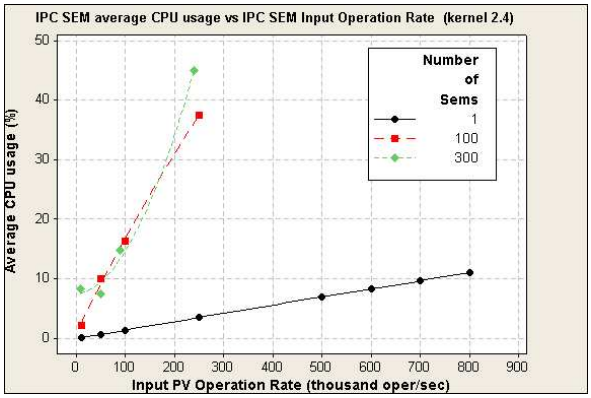


(a) kernel 2.4

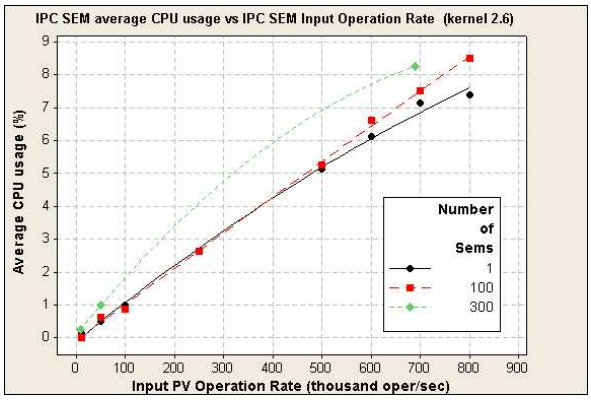


(b) kernel 2.6

Figure 5: Semaphore chart



(a) kernel 2.4



(b) kernel 2.6

Figure 6: CPU usage chart

- Analyze the obtained results to understand whether your hardware can withstand your design.

5 Future Plans for Further Approaches Development

In its current version, the PAC tool allows us to reach the goals we set for ourselves, and has a lot of potential opportunities for further improvements. We realize a number of areas where we can improve this solution to make it more and more applicable for a variety of performance- and availability-related testing.

Through real applications of the PAC tool to existing telecommunication platforms, we realized that this approach from the experimental perspective could be very fruitful. However, we noticed that results may vary depending on the overall understanding and intuition of the person performing the planning of the experiments. If a person using PAC does not spend enough time to investigate the nature of the system, she/he may need to spend several cycles of experiments-planning before she/he identifies right interval of system parameters—i.e., where to search for valuable statistical results. This is still valuable, but requires the boring sorting of a significant amount of statistical information. In reality, there may be more than one hundred tunable parameters. Some of them will not have any impact on certain system characteristics; others will certainly have impact. It is not always easy to predict it just from common perspective. An even more difficult task is to imagine and predict the whole complexity of the inter-relationships of parameters. Automation of this process seems to be reasonable here and there is a math theory devoted to this task that we believe could be successfully applied.

We are talking about multi-parametric optimization. It is a well described area of mathematics, but many constraints are applied to make this theory applicable for discrete and non-linear dependencies (which is true for most of dependencies we can meet in system tunable parameters area). We are currently looking for numeric approaches for these kinds of multi-parametric optimizations—e.g., NOMAD (Nonlinear

Optimization for Mixed variables and Derivatives) [1], GANSO (Global And Non-Smooth Optimization) [7], or ParEGO Hybrid algorithm [14]. A direct search for the optimal parameters combination would take too much time (many years) to sort out all possible combinations, even with fewer than one hundred parameters. Using math theory methods, we will cut the number of experiments down to a reasonable number and shorten the test cycle length in the case of using PAC for load and stress testing purposes.

Our discussion in the previous paragraph is about performance tuning, but it is not the only task that we perform using the PAC tool. Another important thing where the PAC tool is very valuable is the investigation of performance and availability issues. Currently, we perform

	Interface	Automation	Math functionality	Stability	Compatibility	Flexibility	Project activity
Kst	7	5	5	7	8	8	8
Grace	6	7	8	7	8	6	7
LabPlot	8	2	7	7	8	8	7
KDEedu (KmPlot)	7	2	2	7	2	2	8
Metagraf-3D	-	-	-	2	-	-	6
GNUPLOT	6	7	5	8	4	6	4
Root	6	7	8	8	8	8	9

Table 2: Visualization tools

analysis of the results received manually through the use of packages such as MiniTAB, which in many case is time consuming. Our plan is to incorporate statistical analysis methods in the PAC tool in order to allow it to generate statistical analysis reports and to perform results visualization automatically by using GNU GSL or similar packages for data analysis, and such packages as GNUPLOT, LabPlot, or Root for visualization [9][6].

6 Conclusion

In this paper, we have provided an overview of specific performance and availability challenges encountered in Linux servers running on telecommunication networks, and we demonstrated a strong correlation between these challenges and the current trend from Linux vendors to focus on improving the performance and availability of the Linux kernel.

We have briefly described the existing means to address basic performance and availability problem areas, and presented the reason why in each particular case the set of tools used should be different, as well as mentioned that in general the procedure of performance and availability characterization is very time- and resource-consuming.

We presented on the need to have a common integrated approach, i.e., the PAC tool. We discussed the tool architecture and used examples that significantly simplify and unify procedures of performance and availability characterization and may be used in any target problem areas starting from Linux platform parameters tuning, and finishing with load/stress testing and system behavior modeling.

7 Acknowledgements

We would like to the members of the PAC team—namely Sergey Satskiy, Denis Nikolaev, and Alexander Velikotny—for their substantial contributions to the design and development of the tool. We appreciate the review and feedback from of Ibrahim Haddad from Motorola Software Group. We are also thankful to the team of students from Saint-Petersburg State Polytechnic University for their help in investigating possible approaches to information characterization. Finally, we would like to offer our appreciation to Mikhail Chernorutsky, Head of Telecommunication Department at Motorola Software Group in Saint-Petersburg, for his contributions in facilitating and encouraging the work on this paper.

Ed. Note: the Formatting Team thanks Máirín Duffy for creating structured graphics versions of Figures 1, 2, 3, and 4 with Inkscape.

References

[1] Charles Audet and Dominique Orbany. Finding optimal algorithmic parameters using a mesh adaptive direct search, 2004. `http://www.optimization-online.org/DB_HTML/2004/12/1007.html`.

[2] Alberto Avritzer, Joe Kondek, Danielle Liu, and Elaine J. Weyuker. Software Performance Testing Based on Workload Characterization. In *Proceedings of the 3rd international workshop on Software and performance*, 2002. `http://delivery.acm.org/10.1145/590000/584373/p17-avritzer.pdf`.

[3] Steve Best. *Linux®Debugging and Performance Tuning: Tips and Techniques*. Prentice Hall PTR, 2005. ISBN 0-13-149247-0.

[4] Aaron B. Brown. A Decompositional Approach to Computer System Performance Evaluation, 1997. http://www.flashsear.net/fs/prof/papers/harvard-thesis-tr03-97.pdf.

[5] Aaron B. Brown. Towards Availability and Maintainability Benchmarks: A Case Study of Software RAID Systems. Computer science division technical report, UC Berkeley, 2001. http://roc.cs.berkeley.edu/papers/masters.pdf.

[6] Rene Brun and Fons Rademakers. ROOT - An Object Oriented Data Analysis Framework. In *Proceedings of AIHENP conference in Laussanne*, 1996. http://root.cern.ch/root/Publications.htm.

[7] CIAO. *GANSO. Global And Non-Smooth Optimization*. School of Information Technology and Mathematical Sciences, University of Ballarat, 2005. Version 1.0 User Manual. http://www.ganso.com.au/ganso.pdf.

[8] Christian Engelmann and Stephen L. Scott. Symmetric Active/Active High Availability for High-Performance Computing System Services. *JOURNAL OF COMPUTERS*, December 2006. http://www.academypublisher.com/jcp/vol01/no08/jcp01084354.pdf.

[9] Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Michael Booth, and Fabrice Rossi. *GNU Scientific Library*. Free Software Foundation, Inc., 2006. Reference Manual. Edition 1.8, for GSL Version 1.8. http://sscc.northwestern.edu/docs/gsl-ref.pdf.

[10] Ibrahim Haddad. Open-Source Web Servers: Performance on a Carrier-Class Linux Platform. *Linux Journal*, November 2001. http://www2.linuxjournal.com/article/4752.

[11] Ibrahim Haddad. Linux and Open Source in Telecommunications. *Linux Journal*, September 2006. http://www.linuxjournal.com/article/9128.

[12] Daniel Haggander and Lars Lundberg. Attacking the Dynamic Memory Problem for SMPs. In *the 13th International Conference on Parallel and Distributed Computing System (PDCS'2000)*. University of Karlskrona/Ronneby Dept. of Computer Science, 2000. http://www.ide.hk-r.se/~dha/pdcs2.ps.

[13] Mary Jander. Will telecom love Linux?, 2002. http://www.networkworld.com/newsletters/optical/01311301.html.

[14] Joshua Knowles. ParEGO: A Hybrid Algorithm with On-line Landscape Approximation for Expensive Multiobjective Optimization Problems. In *Proceedings of IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, VOL. 10, NO. 1*, 2005. http://ieeexplore.ieee.org/iel5/4235/33420/01583627.pdf.

[15] Jouni Korhonen. Performance Implications of the Multi Layer Mobility in a Wireless Operator Networks, 2004. http://www.cs.helsinki.fi/u/kraatika/Courses/Berkeley04/KorhonenPaper.pdf.

[16] Rick Lehrbaum. Embedded Linux Targets Telecom Infrastructure. *LINUX Journal*, May 2002. http://www.linuxjournal.com/article/5850.

[17] Karim Mattar, Ashwin Sridharan, Hui Zang, Ibrahim Matta, and Azer Bestavros. TCP over CDMA2000 Networks : A Cross-Layer Measurement Study. In *Proceedings of Passive and Active Measurement Conference (PAM 2007)*. Louvain-la-neuve, Belgium, 2007. http://ipmon.sprint.com/publications/uploads/1xRTT_active.pdf.

[18] Kiran Nagaraja, Neeraj Krishnan, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Quantifying and Improving the Availability of High-Performance Cluster-Based Internet Services. In *Proceedings of the ACM/IEEE SC2003 Conference (SCŠ03)*. ACM, 2003. http://ieeexplore.ieee.org/iel5/10619/33528/01592930.pdf.

[19] Neeraj Nehra, R.B. Patel, and V.K. Bhat. A Framework for Distributed Dynamic Load Balancing in Heterogeneous Cluster. *Journal of Computer Science v3*, 2007. http://www.scipub.org/fulltext/jcs/jcs3114-24.pdf.

[20] Nortel. Nortel MSC Server. The GSM-UMTS MSC Server, 2006. http://www.nortel.com/solutions/wireless/collateral/nn117640.pdf.

[21] Hong Ong, Rajagopal Subramaniyan, and R. Scott Studham. Performance Implications of the Multi Layer Mobility in a Wireless Operator Networks, 2005. Performance Modeling and Application Profiling Workshop, SDSC,http://xcr.cenit.latech.edu/wlc/papers/openwlc_sd_2005.pdf.

[22] Sameer Shende, Allen D. Malony, and Alan Morris. Workload Characterization using the TAU Performance System, 2007. http://www.cs.uoregon.edu/research/paracomp/papers/talks/para06/para06b.pdf.

[23] Charles P. Wright, Nikolai Joukov, Devaki Kulkarni, Yevgeniy Miretskiy, and Erez Zadok. Towards Availability and Maintainability Benchmarks: A Case Study of Software RAID Systems. In *proceedings of the 2005 Annual USENIX Technical Conference, FREENIX Track*, 2005. http://www.am-utils.org/docs/apv2/apv2.htm.

"Turning the Page" on Hugetlb Interfaces

Adam G. Litke
IBM
agl@us.ibm.com

Abstract

HugeTLBFS was devised to meet the needs of large database users. Applications use filesystem calls to explicitly request superpages. This interface has been extended over time to meet the needs of new users, leading to increased complexity and misunderstood semantics. For these reasons, hugeTLBFS is unsuitable for potential users like HPC, embedded, and the desktop. This paper will introduce a new interface abstraction for superpages, enabling multiple interfaces to coexist, each with separate semantics.

To begin, a basic introduction to virtual memory and how page size can affect performance is described. Next, the differing semantic properties of common memory object types will be discussed with a focus on how those differences relate to superpages. A brief history of hugeTLBFS and an overview of its design is presented, followed by an explanation of some of its problems. Next, a new abstraction layer that enables alternative superpage interfaces to be developed is proposed. We describe some extended superpage semantics and a character device that could be used to implement them. The paper concludes with some goals and future work items.

1 Introduction

Address translation is a fundamental operation in virtual memory systems. Virtual addresses must be converted to physical addresses using the system page tables. The *Translation Lookaside Buffer (TLB)* is a hardware cache that stores these translations for quick retrieval. While system memory sizes increase exponentially, the TLB has remained small. *TLB coverage*, the percent of total virtual memory that can be translated into physical addresses directly through the cache, has decreased by a factor of 100 in the last ten years [4]. This has resulted in a greater number of *TLB misses* and reduced system performance.

This paper will discuss extensions provided by hardware which can serve to alleviate TLB coverage issues. Properly leveraging these extensions in the operating system is challenging due to the persistent trade-offs between code complexity, performance benefits, and the need to maintain system stability. We propose an extensible mechanism that enables TLB coverage issues to be solved without adverse effects.

2 Hardware Management

Virtual memory is a technique employed by most modern computer hardware and operating systems. The concept can be implemented in many ways but this paper focuses on the *Linux virtual memory manager (VM)*. The physical memory present in the system is divided into equally-sized units called *page frames*. Memory within these page frames can be referred to by a hardware-assigned location called a *physical address*. Only the operating system kernel has direct access to page frames via the physical address. Programs running in user mode must access memory through a *virtual address*. This address is software-assigned and arbitrary. The VM is responsible for establishing the mapping from virtual addresses to physical addresses so that the actual data inside of the page frame can be accessed. In addition to address translation, the VM is responsible for knowing how each page frame is being used. Quite a few data structures exist for tracking this information. A *page table entry (PTE)*, besides storing a physical address, lists the access permissions for a page frame. The assigned permissions are enforced at the hardware level. A *struct page* is also maintained for each page frame in the system [2]. This structure stores status information such as the number of current users and whether the page frame is undergoing disk I/O.

When a program accesses a block of memory for the first time, a translation will not be available to the CPU so control is passed to the VM. A page frame is allocated

for the process and a PTE is inserted into the *page tables*. The TLB is filled with the new virtual to physical translation and execution continues. Subsequent references to this page will trigger a TLB look-up in hardware. A *TLB hit* occurs if the translation is found in the cache and execution can continue immediately. Failure to find the translation is called a *TLB miss*. When a TLB miss occurs, a significant amount of overhead is incurred. The page tables must be traversed to find the page frame which results in additional memory references.

The TLB is conceptually a small array, each slot containing translation information for one page of memory. Over time it has remained small, generally containing 128 or fewer entries [6]. On a system where the page size is 4 KiB, a TLB with 128 slots could cache translations for 512 KiB of memory. This calculation provides a measure of *TLB reach*. Maximizing TLB reach is a worthy endeavor because it will reduce TLB misses.

The increasing complexity of today's applications require large working sets. A *working set* is the smallest collection of information that must be present in main memory to ensure efficient operation of the program [1]. When TLB reach is smaller than the working set, a problem may arise for programs that do not exhibit *locality of reference*. The principle states that data stored in the same place is likely to be used at the same time. If a large working set is accessed sparsely, the limited number of TLB entries will suffer a cycle of eviction and refilling called *TLB thrashing*.

Scientific applications, databases, and various other workloads exhibit this behavior and the resulting marginalization of the TLB. One way to mitigate the problem is to increase TLB reach. As a simple product, it can be increased either by enlarging the TLB or by increasing the page size. Hardware vendors are addressing the issue by devoting more silicon to the TLB and by supporting the use of an increasing number of page sizes. It is up to the operating system to leverage these hardware features to realize the maximum benefits.

2.1 Superpages

Superpages is the term used to refer to any page with a size that is greater than the base page size. Significant research has been done to develop algorithms for, and

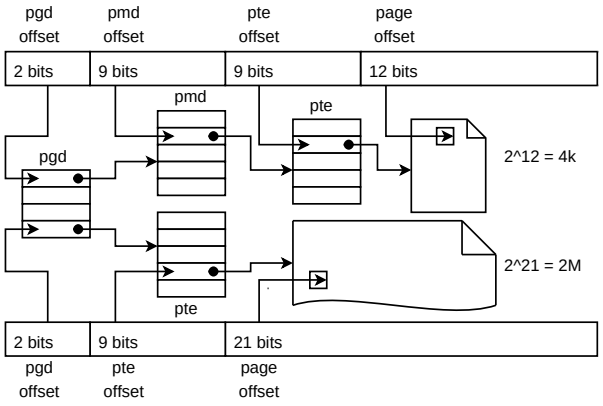


Figure 1: Example page table structure given different page sizes

measure the performance of superpages. Some workloads such as scientific applications and large databases commonly see gains between 10 and 15 percent when using superpages. Gains as high as 30% have been measured in some cases [5]. Superpages are ideal for an application that has a persistent, large, densely accessed working set. The best candidates tend to be computationally intensive.

Virtual memory management with superpages is more complex. Due to the way a virtual address indexes the page table structures and the target page, multiple page sizes necessitate multiple page table formats. Figure 1 contrasts the page table layout for two different page sizes using the x86 architecture as an example. As the figure shows, a virtual address is divided into a series of offsets. The page offset must use enough bits to reference every byte in the page to which the virtual address refers. A 4 KiB page requires 12 bits but a 2 MiB page needs 21. Since, in this example, the virtual address is only 32 bits, supporting a big page offset necessitates changes to the other offsets. In this example, the pmd page table level is simply removed which results in a two level page table structure the 2 MiB pages.

For most workloads, superpages have no effect or may actually hinder performance. Due to *internal fragmentation*, larger page sizes can result in wasted memory and additional, unnecessary work for the operating system. For example, to satisfy security requirements a newly allocated page must be filled with zeroes before it is given to a process. Even if the process only intends to use a small portion of the page, the entire page must be allocated and zeroed. This condition is worst for applications that sparsely access their allocated memory.

Although superpages are complex and improve performance only for programs with a specific set of characteristics, where they do help the impact is great. For this reason, any implementation of superpages should be flexible to maximize performance potential while placing as small of a burden on the system as possible.

3 Address Space Management

By insulating the application from the physical memory layout, the kernel can regulate memory usage to improve security and performance. Programs can be guaranteed private memory allowing sharing only under controlled circumstances. Optimizations, such as automatically sharing common read-only data among processes is made possible. This leads to memory with different semantic characteristics, two of which are particularly relevant when discussing superpages and their implementation in Linux.

3.1 Shared Versus Private

A block of memory can be either shared or private to a process. When memory is being accessed in shared mode, multiple processes can read and, depending on access permissions, write to the same set of pages which are shared among all the attached processes. This means that modifications made to the memory by one process will be seen by every other process using that memory. This mode is clearly useful for things such as interprocess communication.

Memory is most commonly accessed with private semantics. A private page appears exclusive to one address space and therefore only one process may modify its contents. As an optimization, the kernel may share a private page among multiple processes as long as it remains unmodified. When a write is attempted on such a page it must first be unshared. This is done by creating a new copy of the original page and permitting changes only to the new copy. This operation is called *copy on write (COW)* [2].

3.2 File-backed Versus Anonymous

The second semantic characteristic concerns the source of the data that occupies a memory area. Memory can be either file-backed or anonymous. File-backed memory is essentially an in memory cache of file data from a disk or other permanent storage. When pages of this memory type are accessed, the virtual memory manager transparently performs any disk I/O necessary to ensure the in memory copy of the data is in sync with the master version on disk. Maintaining the coherency of many copies of the same data is a significant source of complexity in the Linux memory management code.

Anonymous memory areas generally have no associated backing data store. Under memory pressure, the data may be associated with a swap file and written out to disk, but this is not a persistent arrangement as with file-backed areas. When pages are allocated for this type of memory, they are filled with zeroes.

Linux is focused on furnishing superpage memory with anonymous and shared semantics. This means that superpages can be trivially used with only a small subset of the commonly used memory objects.

3.3 Memory Objects

Memory is used by applications for many different purposes. The application places requirements on each area, for example the code must be executable and the data writable. The operating system enforces additional constraints, such as preventing writes to the code or preventing execution of the stack. Together these define the required semantics for each area.

The stack is crucial to the procedural programming model. It is organized into *frames*, where each frame stores the local variables and return address for a function in the active function call chain. Memory in this area is private and anonymous and is typically no larger than a few base pages. The access pattern is localized since executing code tends to only access data in its own frame at the top of the stack. It is unlikely that such a small, densely accessed area would benefit from superpages.

Another class of objects are the memory resident components of an executable program. These can be further divided into the *machine executable code (text)* and the *program variables (data)*. These objects use file-backed, private memory. The access pattern depends heavily on the specific program, but when they are sufficiently large, superpages can provide significant performance gains. The text and data segments of shared libraries are handled slightly different but have similar semantic properties to executable program segments.

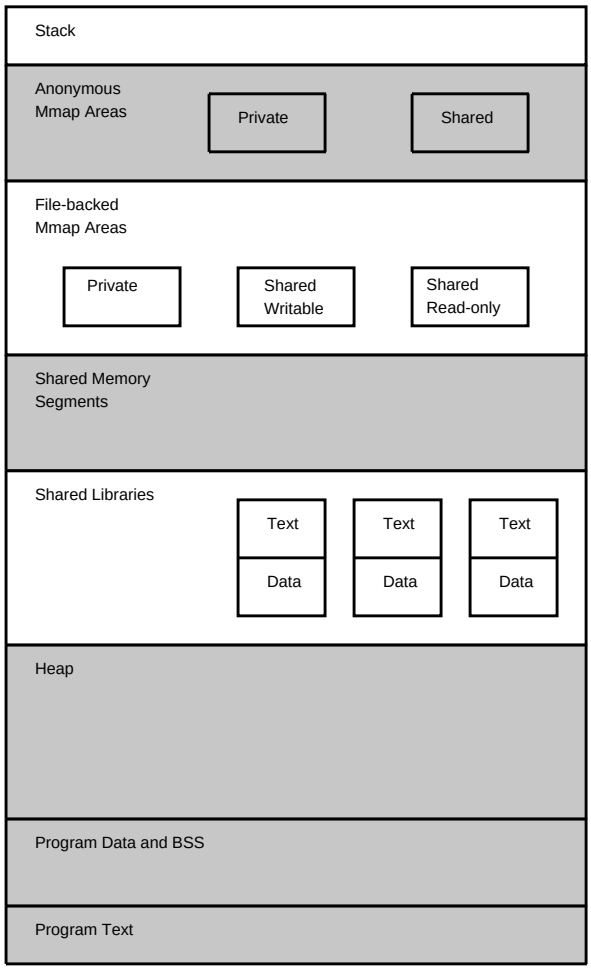


Figure 2: Memory object types

The *heap* is used for dynamic memory allocation such as with malloc. It is private, anonymous memory and can be fairly trivially backed with superpages. Programs that use lots of dynamic memory may see performance gains from superpages.

Shared memory segments are used for interprocess communication. Processes attach them by using a global shared memory identifier. Database management systems use large shared memory segments to cache databases in memory. Backing that data with superpages has demonstrated substantial benefits and inspired the inclusion of superpage support in Linux.

It is already possible to use superpages for some memory types directly. Shared memory segments and special, explicit memory maps are supported. By using libraries, superpages can be extended in a limited way to a wider variety of memory objects such as the heap and program segments.

4 Introduction to HugeTLBFS

In 2002, developers began posting patches to make superpages available to Linux applications. Several approaches were proposed and, as a result of the ensuing discussions, a set of requirements evolved. First and foremost, the existing VM code could not be unduly complicated by superpage support. Second, databases were the application class most likely to benefit from superpage usage at the time, so the interface had to be suitable for them. *HugeTLBFS* was deemed to satisfy these design requirements and was merged at the end of 2002.

HugeTLBFS is a RAM-based filesystem. The data it contains only exists within superpages in memory. There is no backing store such as a hard disk. HugeTLBFS supports one page size called a *huge page*. The size of a huge page varies depending on the architecture and other factors. To access huge page backed memory, an application may use one of two methods. A file can be created and `mmap()`ed on a mounted hugeTLBFS filesystem, or a specially created shared memory segment can be used. HugeTLBFS continues to serve databases well, but other applications use memory in different ways and find this mechanism unsuitable.

The last few years have seen a dramatic increase in enthusiasm for superpages from the scientific community. Researchers run multithreaded jobs to process massive amounts of data on fast computers equipped with huge amounts of memory. The data sets tend to reside in portions of memory with private semantics such as the *BSS* and heap. Heavy usage of these memory areas is a general characteristic of the *Fortran* programming language. To accommodate these new users, private mapping support was added to hugeTLBFS in early 2006. *LibHugeTLBFS* was written to facilitate the remapping of executable segments and heap memory into huge pages. This improved performance for a whole new class of applications, but for a price.

For shared mappings, the huge pages are reserved at creation time and are guaranteed to be available. Private mappings are subject to non-deterministic COW operations which make it impossible to determine the number of huge pages that will actually be needed. For this reason, successful allocation of huge pages to satisfy a private mapping is not guaranteed. Huge pages are a scarce resource and they frequently run out. If this happens while trying to satisfy a huge page fault, the application

will be killed. This unfortunate consequence makes the use of private huge pages unreliable.

The data mapped into huge pages by applications is accessible in files via globally visible mount points. This makes it easy for any process with sufficient privileges to read, and possibly modify, the sensitive data of another process. The problem can be partially solved through careful use of multiple hugeTLBFS mounts with appropriate permissions. Despite safe file permissions, an unwanted covert communication channel could still exist among processes run by the same user.

HugeTLBFS has become a mature kernel interface with countless users who depend on consistency and stability. Adapting the code for new superpage usage scenarios, or even to fix the problems mentioned above has become difficult. As hardware vendors make changes to solve TLB coverage issues such as adding support for multiple superpage sizes, Linux is left unable to capitalize due to the inflexibility of the hugeTLBFS interface.

For a real example of these problems, one must only look back to the addition of *demand faulting* to hugeTLBFS. Before enablement of this feature, huge pages were always *prefaulted*. This means that when huge pages were requested via an `mmap()` or `shmat()` system call, all of them were allocated and installed into the mapping immediately. If enough pages were not available or some other error occurred, the system call would fail right away. Demand faulting delays the allocation of huge pages until they are actually needed. A side effect of this change is that huge page allocation errors are delayed along with the faults.

A commercial database relied on the strict accounting semantics provided by prefault mode. The program mapped one huge page at a time until it failed. This effectively reserved all available huge pages for use by the database manager. When hugeTLBFS switched to demand faulting, the `mmap()` calls would never fail so the algorithm falsely assumed an inflated number of huge pages were available and reserved. Even with a smarter huge page reservation algorithm, the database program could be killed at a non-deterministic point in the future when the huge page pool is exhausted.

5 Page Table Abstraction

HugeTLBFS is complex and its semantics are rigid. This makes it an unsuitable vehicle for future development. To quantify the potential benefits of expanded superpage usage, a mechanism for testing competing and potentially incompatible semantics and interfaces is needed. Making this possible requires overriding the superpage-related special cases that are currently hardwired to hugeTLBFS.

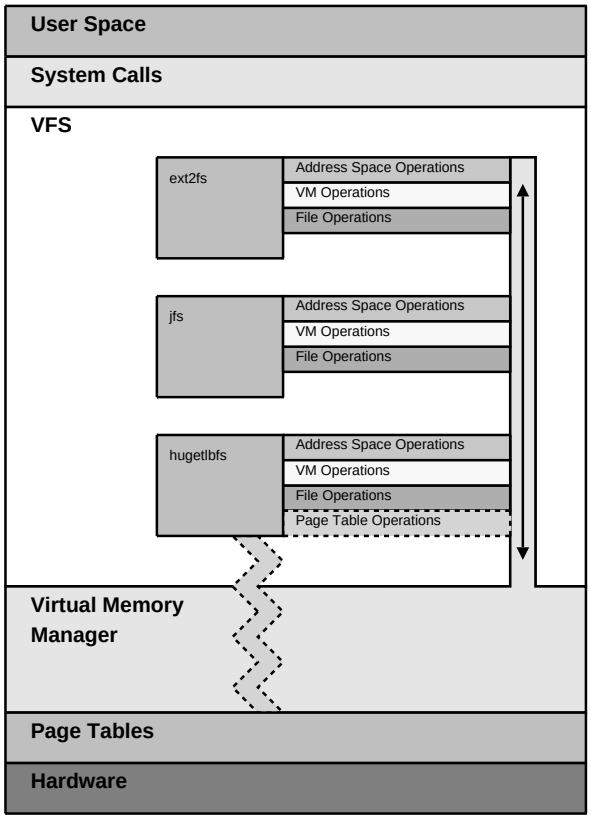


Figure 3: How hugeTLBFS interacts with the rest of the kernel

In finding a solution to this problem, it is important to remember the conditions that grounded the development of hugeTLBFS. Additional complexity cannot be added to the VM. Transparent, fully-integrated superpage support is thus an unrealistic pursuit. As with any kernel change, care must be taken to not disturb existing users. The behavior of hugeTLBFS must not change and performance regressions of any kind must be avoided. Further, a complete solution must fully abstract existing special cases but remain extensible.

Like any other filesystem, hugeTLBFS makes use of an abstraction called the *virtual filesystem (VFS)* layer. This object-oriented interface is what makes it possible for the VM to consistently interact with a diverse array of filesystems. Many of the special cases introduced by supporting superpages are hidden from the VM through

hugeTLBFS' effective use of the VFS API. This interface was not designed to solve page size issues so some parts of the kernel remain hugeTLBFS-aware. Most of these cases are due to the need for an alternate page table format for superpages. Figure 3 shows how hugeTLBFS fits into the VFS layer as compared to other filesystems.

Taking the VFS layer as inspiration, we propose an abstraction for page table operations that is capable of unwiring the remaining hugeTLBFS-specific hooks. It is simply a structure of six function pointers that fully represent the set of special page table operations needed.

`fault()` is called when a page fault occurs somewhere inside a VMA. This function is responsible for finding the correct page and instantiating it into the page tables.

`copy_vma()` is used during fork to copy page table entries from a VMA in the parent process to a VMA in the child process.

`change_protection()` is called to modify the protection bits of PTEs for a range of memory.

`pin_pages()` is used to instantiate the pages in a range of userspace memory. The kernel is not generally permitted to take page faults. When accessing userspace, the kernel must first ensure that all pages in the range to be accessed are present.

`unmap_page_range()` is needed when unmapping a VMA. The page tables are traversed and all instantiated pages for a given memory range are unmapped. The PTEs for the covered range are cleared and any pages which are no longer in use are freed.

`free_pgtable_range()` is called after all of the PTEs in a mapping are cleared to release the pages that were used to store the actual page tables.

5.1 Evaluating the Solution

The page table operations produce a complete and extensible solution. All the infrastructure is in place to enable hugeTLBFS to be built as a module, further separating it from the core VM. Other independent superpage interface modules can be added to the system without causing interference.

The implementation is simple. The existing, hugeTLBFS page table manipulation functions are collected into an operations structure without modifying them in any way. Instead of calling hard-wired functions, the function to call is looked up in the page table operations. Changing only these two elements ensures that hugeTLBFS behavior is unchanged.

Superpages exist solely for performance reasons so an implementation that is inefficient serves only to undermine its own existence. Two types of overhead are considered with respect to the proposed changes. One pointer will be added to the VMA. Already, this structure does not fit into a cache line on some CPUs, so care must be taken to place the new field at a sensible offset within the structure definition. To this end, the `pagetable_ops` pointer has been placed near the `vm_ops` and `vm_flags` fields, which are also used frequently when manipulating page tables.

The second performance consideration is related to the pointer indirection added when a structure of function pointers is used. Instead of simply jumping to an address that can be determined at link time, a small amount of additional work is required. The address of the assigned operations is looked up in the VMA. This address, plus an offset, is dereferenced to yield the address of the function to be called. VMAs that do not implement the page table operations do not incur any overhead. Instead of checking `vm_flags` for `VM_HUGETLB`, `pagetable_ops` is checked for a NULL value.

6 Using the Abstraction Interface

With a small amount of simple abstraction code, the kernel has become more flexible and is suitable for further superpage development. HugeTLBFS and its existing users remain undisturbed and performance of the system has not been affected in any measurable way. We now describe some possible applications of this new extensible interface.

A character device is an attractive alternative to the existing filesystem interface for several reasons. Its semantics provide a more natural and secure method for allocating anonymous, private memory. The interface code is far simpler than that of hugeTLBFS which makes it a much more agile base for the following extensions.

The optimal page size for a particular memory area depends on many factors such as: total size, density of

```
struct pagetable_operations_struct page_pagetable_ops = {
     .copy_vma           = copy_hugetlb_page_range,
     .pin_pages          = follow_hugetlb_page,
     .unmap_page_range   = unmap_hugepage_range,
     .change_protection  = hugetlb_change_protection,
     .free_pgtable_range = hugetlb_free_pgd_range,
     .fault              = page_fault,
};
```

Figure 4: A sample page table operations structure

access, and frequency of access. If the page size is too small, the system will have to service more page faults, the TLB will not be able to cache enough virtual to physical translations, and performance will suffer. If the page size is too large, both time and memory are wasted. On the PowerPC® architecture, certain processors can use pages in sizes of 4KiB, 64KiB, 16MiB, and larger. Other platforms can also support more than the two page sizes Linux allows. Architecture specific code can be modified to support more than two page sizes at the same time. Fitted with a mechanism to set the desired size, a character device will provide a simple page allocation interface and make it possible to measure the effects of different page sizes on a wide variety of applications.

Superpages are a scarce resource. Fragmentation of memory over time makes allocation of large, contiguous blocks of physical memory nearly impossible. Without contiguous physical memory, superpages cannot be constructed. Unlike with normal pages, swap space cannot be used to reclaim superpages. One strategy for dealing with this problem is *demotion*. When a superpage allocation fails, a portion of the process' memory can be converted to use normal pages. This allows the process to continue running in exchange for a sacrifice of some of the superpage performance benefits.

Selecting the best page size for the memory areas of a process can be complex because it depends on factors such as application behavior and the general system environment. It is possible to let the system choose the best page size based on variables it can monitor. For example, a *population map* can be used to keep track of allocated base pages in a memory region [5]. Densely populated regions could be *promoted* to superpages automatically.

6.1 A Simple Character Device

The character device is a simple driver modeled after `/dev/zero`. While the basic functionality could be implemented via hugeTLBFS, that would subject it to the functional limitations previously described. Additionally, it could not be used to support development of the extensions just described. For these reasons, the implementation uses page table abstraction and is independent of hugeTLBFS.

The code is self contained and can be divided into three distinct components. The first component is the standard infrastructure needed by all drivers. This device is relatively simple and needs only a small function to register with the driver layer.

The second component is the set of device-specific structures that define the required abstracted operations. Figure 4 shows the page table operations for the character device. Huge page utility functions are already well separated from the hugeTLBFS interface which makes code reuse possible. Five of the six page table operations share the same functions used by hugeTLBFS.

The third component is what makes this device unique. It handles page faults differently than hugeTLBFS so a special `fault()` function is defined in the page table operations. This function is simpler than the hugeTLBFS fault handler because it does not need to handle shared mappings or filesystem operations such as truncation. Most of the proposed semantic changes can be implemented by further modifying the fault handler.

7 Conclusions

The page table operations abstraction was developed to enable advanced superpage development and work

around some problems in hugeTLBFS. It does not alter the behavior of the current interface nor complicate the kernel in any significant way. No performance penalty could be measured. Work on the topics previously described can now begin.

The described changes have also led to some general improvements in the way hugeTLBFS interacts with the rest of the kernel. By collecting a set of dispersed hugeTLBFS-specific page table calls into one structure, the list of overloaded operations becomes clear. This API, when coupled with other pending cleanups, removes hard-coded special cases from the kernel. The result is a hugeTLBFS implementation that is even further "on the side" and decoupled from the core memory manager.

8 Future Work

During development of the page table abstraction interface and the character device, a few additional opportunities to clean up the interface between hugeTLBFS and the rest of the kernel became apparent. Every effort should be made to extricate the remaining hugeTLBFS special cases from the kernel. Moving superpage-related logic behind the appropriate abstractions makes for a simpler VM and at the same time enables richer superpage support.

The work presented in this paper enables a substantial body of research and development using Linux. We intend to implement different superpage semantics and measure their performance effects across a variety of workloads and real applications. If good gains can be achieved with reasonable code we hope to see those gains realized outside of our incubator in production kernels.

A separate effort to reduce memory fragmentation is underway [3]. If this body of work makes it into the kernel, it will have a positive effect on the usability of superpages in Linux. When contiguous blocks of physical memory are readily available, superpages can be allocated and freed as needed. This makes them easier to use in more situations and with greater flexibility. For example, page size promotion and demotion become more effective if memory can be allocated directly from the system instead of from the static pool of superpages that exists today.

9 Legal Statement

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] P. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[2] M. Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall Upper Saddle River, NJ, 2004.

[3] M. Gorman and A. Whitcroft. The What, The Why and the Where To of Anti-Fragmentation. *Ottawa Linux Symposium*, 1:369–384, 2006.

[4] J. Navarro. *Transparent operating system support for superpages*. PhD thesis, RICE UNIVERSITY, 2004.

[5] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review*, 36(si):89, 2002.

[6] T. Selén. *Reorganisation in the skewed-associative TLB*. Department of Information Technology, Uppsala University.

Resource Management: Beancounters

Pavel Emelianov
xemul@openvz.org

Denis Lunev
den@openvz.org

Kirill Korotaev
dev@openvz.org

Abstract

The paper outlines various means of resource management available in the Linux kernel, such as per-process limits (the `setrlimit(2)` interface), shows their shortcomings, and illustrates the need for another resource control mechanism: beancounters.

Beancounters are a set of per-process group parameters (proposed and implemented by Alan Cox and Andrey Savochkin and further developed for OpenVZ) which can be used with or without containers.

Beancounters history, architecture, goals, efficiency, and some in-depth implementation details are given.

1 Current state of resource management in the Linux kernel

Currently the Linux kernel has only one way to control resource consumption of running processes—it is UNIX-like resource limits (rlimits).

Rlimits set upper bounds for some resource usage parameters. Most of these limits apply to a single process, except for the limit for the number of processes, which applies to a user.

The main goal of these limits is to protect processes from an **accidental** misbehavior (like infinite memory consumption) of other processes. A better way to organize such a protection would be to set up a **minimal** amount of resources that are **always** available to the process. But the old way (specifying upper limits to catch some cases of excessive consumption) may work, too.

It is clear that the reason for introducing limits was the protection against an accidental misbehavior of processes. For example, there are separate limits for the data and stack size, but the limit on the total memory consumed by the stack, data, BSS, and memory mapped regions does not exist. Also, it is worth to note that the `RLIMIT_CORE` and `RLIMIT_RSS` limits are mostly ignored by the Linux kernel.

Again, most of these resource limits apply to a single process, which means, for example, that all the memory may be consumed by a single user running the appropriate number of processes. Setting the limits in such a way as to have the value of multiplying the per-process limit by the number of processes staying below the available values is impractical.

2 Beancounters

For some time now Linus has been accepting patches adding the so-called namespaces into the kernel. Namespaces allow tasks to observe their own set of particular kernel objects such as IPC objects, PIDs, network devices and interfaces, etc. Since the kernel provides such a grouping for tasks, it is necessary to control the resource consumption of these groups.

The current mainline kernel lacks the ability to track the kernel resource usage in terms of arbitrary task groups, and this ability is required rather badly.

The list of the resources that the kernel provides is huge, but the main resources are:

- memory,
- CPU,
- IO bandwidth,
- disk space,
- Networking bandwidth.

This article describes the architecture, called “beancounters,” which the OpenVZ team proposes for controlling the first resource (memory). Other resource control mechanisms are outside the scope of this article.

2.1 Beancounters history

All the deficiencies of the per-process resource accounting were noticed by Alan Cox and Andrey Savochkin long ago. Then Alan introduced an idea that crucial kernel resources must be handled in terms of groups and set the "beancounter" name for this group. He also stated that once a task moves to a separate group it never comes back and neither do the resources allocated on its requests.

These ideas were further developed by Andrey Savochkin, who proposed the first implementation of beancounters [UB patch]. It included the tracking of process page tables, the numbers of tasks within a beancounter, and the total length of mappings. Further versions included such resources as file locks, pseudo terminals, open files, etc.

Nowadays the beancounters used by OpenVZ control the following resources:

- kernel memory,
- user-space memory,
- number of tasks,
- number of open files and sockets,
- number of PTYs, file locks, pending signals, and iptable rules,
- total size of network buffers,
- active dentry cache size, i.e. the size of the dentries that cannot be shrunk,
- dirty page cache that is used to track the IO activity.

2.2 The beancounters basics

The core object is the beancounter itself. The beancounter represents a task group which is a resource consumer.

Basically a beancounter consists of an ID to make it possible to address the beancounter from user space for changing its parameters and the set of usage-limit pairs to control the usage of several kernel resources.

More precisely, each beancounter contains not just usage-limit pairs, but a more complicated set. It includes the usage, limit, barrier, fail counter, and maximal usage values.

The notion of "barrier" has been introduced to make it possible to start rejecting the allocation of a resources before the limit has been hit. For example when a beancounter hits the mappings barrier, the subsequent `sbrk` and `mmap` calls start to fail, although `execve`, which maps some areas, still works. This allows the administrator to "warn" the tasks that a beancounter is short of resources before the corresponding group hits the limit and the tasks start dying due to unrecoverable rejections of resource allocation.

Allocating a new resource is preceded by "charging" it to the beancounter the current task belongs to. Essentially the charging consists of an atomic checking that the amount of resource consumed doesn't exceed the limit, and adding the resource size to the current usage.

Here is a list of memory resources controlled by the beancounters subsystem:

- total size of allocated kernel space objects,
- size of network buffers,
- lengths of mappings, and
- number of physical pages.

Below are some details on the implementation.

3 Memory management with beancounters

This section describes the way beancounters are used to track memory usage. The kernel memory is accounted separately from the userspace one. First, the userspace memory management is described.

3.1 Userspace memory management

The kernel defines two kinds of requests related to memory allocation.

1. The request to allocate a memory region for physical pages. This is done via the `mmap(2)` system call. The process may only work within a set of

mmap-ed regions, which are called VMAs (virtual memory areas). Such a request does not lead to allocation of any physical memory to the task. On the other hand, it allows for a *graceful reject* from the kernel space, i.e. an error returned by the system call makes it possible for the task to take some actions rather than die suddenly and silently.

2. The request to allocate a physical page within one of the VMAs created before. This request may also create some kernel space objects, e.g. page tables entries. The only way to reject this request is to send a signal—`SEGV`, `BUS`, or `KILL`—depending on the failure severity. This is not a good way of rejecting as not every application expects critical signals to come during normal operation.

The beancounters introduce the "unused page" term to indicate a page in the VMA that has not yet been touched, i.e. a page whose VMA has already been allocated with `mmap`, but the page itself is not yet there. The "used" pages are physical pages.

Kernel VMAs may be classified as:

- reclaimable VMAs, which are backed by some file on the disk. For example, when the kernel needs some more memory, it can safely push the pages from this VMA to the disk with almost no limitation.
- unreclaimable VMAs, which are not backed by any file and the only way to free the pages within such VMAs is push the page to the swap space. This way has certain limitations in that the system may have no swap at all or the swap size may be too small. The reclamation of pages from this kind of VMAs is more likely to fail in comparison with the previous kind.

In the terms defined above, the OpenVZ beancounters account for the following resources:

- the number of used pages within all the VMAs,
- the sum of unused and used pages within unreclaimable VMAs and used pages in reclaimable VMAs.

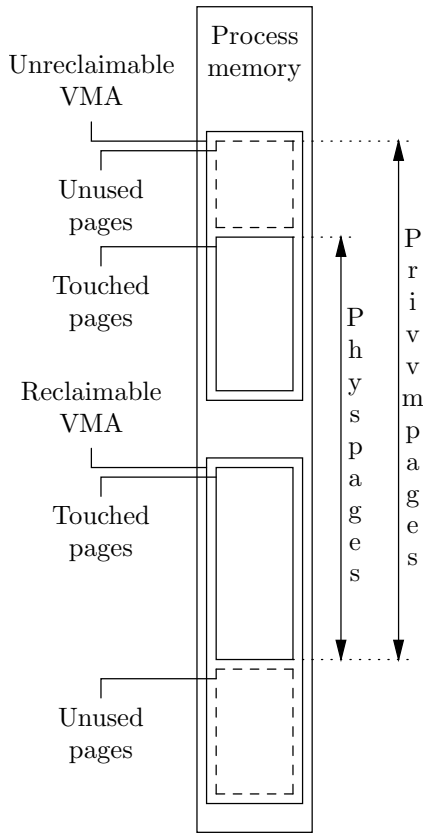


Figure 1: User space memory management

The first resource is account-only (i.e. not limited) and is called "physpages." The second one is limited in respect of only unused pages allocation and is called "privvmpages." This is illustrated in Figure 1.

The only parameter that remains unlimited—the size of pages touched from a disk file—does not create a security hole since the size of files is limited by the OpenVZ disk quotas.

3.2 Physpages management

While privvmpages accounting works with whole pages, physpages accounts for page fractions in the case some pages are shared among beancounters [RSS]. This may happen if a task changes its beancounter or if tasks from different groups map and use the same file.

This is how it works. There is a many-to-many dependence between the mapped pages and the beancounters in the system. This dependence is tracked by a ring of *page beancounters* associated with each mapped page (see Figure 2). What is important is that each page has

its associated circular list of page beancounters and the head of this list has a special meaning.

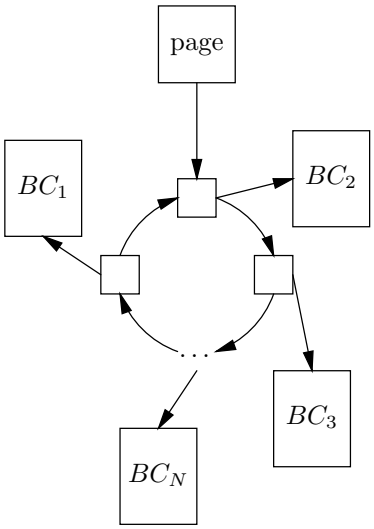


Figure 2: Page shared among beancounters

Naturally, if a page is shared among N beancounters, each beancounter is to get a $\frac{1}{N}$-th part of it. This approach would be the most precise, but also the worst, because adding a page to a new beancounter would require an update of all the beancounters among which the page is already shared.

Instead of doing that, parts of the page equal to

$$\frac{1}{2^{shift(page,beancounter)}}$$

are charged, where $shift(page, beancounter)$ is calculated for

$$\sum_{beancounters} \frac{1}{2^{shift(page,beancounter)}} = 1$$

to be true for each page.

When mapping a page into a new beancounter, half of the part charged to the head of the page's beancounters list is moved to the new beancounter. Thus when the page is sequentially mapped to 4 different beancounters, its fractions would look like

	bc_1	bc_2	bc_3	bc_4
1	1			
2	$\frac{1}{2}$	$\frac{1}{2}$		
3	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	
4	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

When unmapping a page from a beancounter, the fraction of the page charged to this beancounter is returned to one or two of the beancounters on the page list.

For example, when unmapping the page from bc_4 the fractions would change like

$$\{\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4}\} \rightarrow \{\frac{1}{2},\frac{1}{4},\frac{1}{4}\}$$

i.e. a fraction from bc_4 was added to only one beancounter—bc_3.

Next, when unmapping the page from bc_3 with fraction of $\frac{1}{2}$ its charge will be added to two beancounters to keep fractions be the powers of two:

$$\{\frac{1}{2},\frac{1}{4},\frac{1}{4}\} \rightarrow \{\frac{1}{2},\frac{1}{2}\}$$

With that approach the following has been achieved:

- algorithm of adding and removing references to beancounters has $O(1)$ complexity;
- the sum of the physpages values from all the beancounters is the number of RAM pages actually used in the kernel.

3.2.1 Dirty page cache management

The above architecture is used for dirty page cache and thus IO accounting.

Let's see how IO works in the Linux kernel [RLove]. Each page may be either *clean* or *dirty*. Dirty pages are marked with the bit of the same name and are about to be written to disk. The main point here is that dirtying a page doesn't imply that it will be written to disk immediately. When the actual writing starts, the task (and thus the beancounter) that made the page dirty may already be dead.

Another peculiarity of IO is that a page may be marked as dirty in a context different from the one that really owns the page. Arbitrary pages are unmapped when the kernel shrinks the page cache to free more memory. This is done by checking the `pte` dirty flag set by a CPU.

Thus it is necessary to save the context in which a page became dirty until the moment the page becomes clean,

i.e. its data is written to disk. To track the context in question, an IO beancounter is added between the page and its page beancounters ring (see Figure 3). This IO beancounter keeps the appropriate beancounter while the page stays dirty.

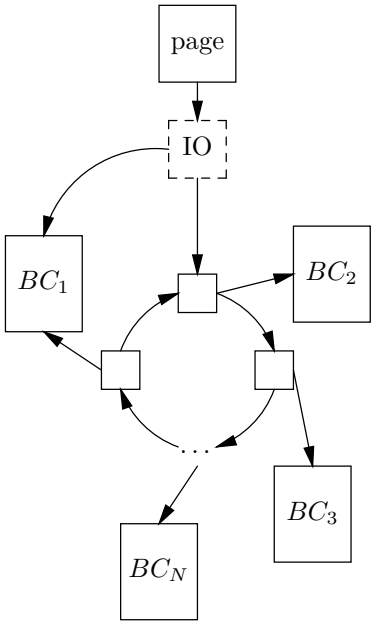


Figure 3: Tracking the dirtier of the page

To determine the context of a page, the page beancounter ring mentioned above is used. When a page becomes dirty in a synchronous context, e.g. a regular `write` happens, the current beancounter is used. When a page becomes dirty in an arbitrary context, e.g. from a page shrinker, the first beancounter from this ring is used.

3.3 Kernel memory management

Privvmpages allow a user to track the memory usage of tasks and give the applications a chance for a graceful exit rather than being killed, but they do not provide any protection. Kernel memory size—the "kmemsize" in the beancounters terms—is the way to protect the system from a DoS attack from userspace.

The kernel memory is a scarce resource on 32-bit systems due to the limited normal zone size. But even on 64-bit systems, applications causing the kernel to consume the unlimited amount of RAM can easily DoS it.

There is a great difference between the user memory and the kernel memory which results in dramatic difference in implementation.

When a user page is freed, the task and thus the beancounter it belongs to is always well known. When a kernel space object is freed, the beancounter it belongs to is almost never known due to refcounting mechanism and RCU. Thus each kernel object should carry a pointer on the beancounter it was allocated by.

The way beancounters store this pointer differs between different techniques of allocating objects.

There are three mechanisms for object allocation in the kernel:

1. *buddy page allocator* – the core mechanism used directly and by the other two. To track the beancounter of the directly allocated page there is an additional pointer on `struct page`. Another possible way would be to allocate a mirrored to `mem_map` array of pointers, or reuse `mapping` field on `struct page`.
2. *vmalloc* – To track the owner of vmalloc-ed object the owner of the first (more precisely—the zeroth) page is used. This is simple as well.
3. *kmalloc* – the way to allocate small objects (down to 32 bytes of size). Many objects allocated with kmalloc must be tracked and thus have a pointer to the corresponding beancounter. First versions of beancounters changed such objects explicitly by adding `struct beancounter *owner` field in structures. To unify this process in new versions, `mm/slab.c` is modified by adding an array of pointers on beancounters at the tail of each slab (see Figure 4). This array is indexed with the sequence number of the object on the slab.

Such an architecture has two potential side effects:

1. slabs will become shorter, i.e. one slab will carry less objects than it did before; and
2. slabs may become "offslab" as the gap will exceed the offslab border.

Table 1 shows the results for `size-X` caches. As seen from this table less than 1% of available objects from "on-page" slabs are lacking. "Offslab" caches do not lack any objects and no slabs become offslab.

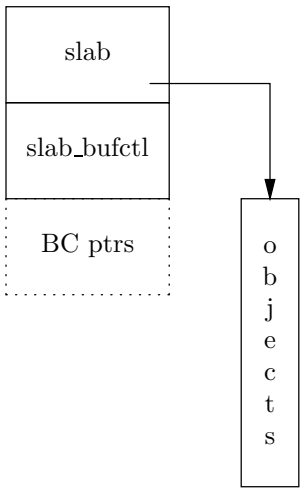


Figure 4: Extending a slab for the kernel memory tracking

Slab size	# of objects		offslab-ness	
	before	after	before	after
size-32	113	101	−	−
size-64	59	56	−	−
size-128	30	29	−	−
size-256	15	15	−	−
size-512	8	8	+	+
size-1024	4	4	+	+
size-2048	2	2	+	+
size-4096 ...	1	1	+	+

Table 1: The comparison of size-X caches with and without beancounters patch

3.4 Network buffers management

Network buffers management [AFTM] fundamentally differs for the send and receive buffers. Basically, the host does not control the incoming traffic and fully control the outgoing traffic.

Unfortunately, the mainstream kernel socket accounting mechanism cannot be reused, as it has known deficiencies:

- slab overhead is not included into the accounted packet size. For Ethernet the difference is around 25%–30% as `size-2048` slab is used for packets. Unfortunately, `skb->truesize` calculations can't be changed without massive TCP/IP stack fix, as this would lead to serious performance degradation due to TCP window reduction; and
- the accounting is not strict, and limits can be overused.

3.4.1 Send buffer space

TCPv4 now and TCPv6 with DCCP in the future are accounted separately from all the other outgoing traffic. Only the packets residing in the socket send queue are charged since skb "clones" sent to device for transmission have a very limited lifetime.

Netlink send operations charging is not uniform in respect to send direction. The buffers are charged for user socket only even if they are sent from the kernel. This is fair as, usually, data stream from the kernel to a user is initiated by the user.

Beancounters track memory usage with the appropriate overheads on per-socket basis. Each socket has a guarantee calculated as

$$\frac{limit - barrier}{N_{sockets}},$$

where $N_{sockets}$ is the limit of sockets of appropriate type on the beancounter.

Beancounters don't break a semantics of the `select` system call, i.e. if it returns that the socket can be written to, `write` will send at least one byte. To achieve this goal an additional field for the socket structure had been introduced, namely, `poll_reserve`. So, actual resource usage is shown as

$$\sum_{s \in sockets} (s_{poll\_reserv} + \sum_{skb \in s_{write\_queue}} skb_{size})$$

3.4.2 Receive buffer space

Management of the receive buffers for non-TCP sockets is simple. Incoming packets are simply dropped if the limit is hit. This is normal for raw IP sockets and UDP, as there is no protocol guarantee that the packet will be delivered.

Though, there is a problem with the netlink sockets. In general, the user (`ip` or similar tool) sends a request

to the kernel and starts waiting for an answer. The answer may never arrive as the kernel does not retransmit netlink responses. Though, the practical impact of this deficiency is acceptable.

TCP buffer space accounting is mostly the same except for the guarantee for one packet. The amount equal to $N_{sockets} * max\_advmss$ is reserved for this purpose. Beancounters rely on the generic code for TCP window space shrinking. Though, a better window management policy for all sockets inside a beancounter is being investigated.

4 Performance

The OpenVZ team spent a lot of time improving the performance of the beancounters patches. The following techniques are employed:

- **Pre-charging of resources on a task creation.** When the task later tries to charge a new portion of resource, it may take the desired amount from this reserve. The same technique is used in network buffers accounting, but pre-charge is stored on socket.
- **On-demand accounting.** Per-group accounting is not performed when the overall resource consumption is low enough. When a system becomes low of some resource per-beancounter accounting is turned on and vice-versa—when a system has a lot of free resources, per-beancounter accounting is turned off. Nevertheless this switch is rather slow as it implies seeking for all resources belonging to a beancounter and thus it should happen rarely.

Test name	Vanilla	OVZ	%
Process Creation	7456	7288	97%
Execl Throughput	2505	2490	99%
Pipe Throughput	4071	4084	100%
Shell Scripts	4521	4369	96%
File Read	1051	1041	99%
File Write	1070	1086	101%

Table 2: Unixbench results comparison (p.1)

The results of unixbench test are shown for the following kernels:

Test name	Vanilla	OVZ	%
Process Creation	7456	6788	91%
Execl Throughput	2505	2290	91%
Pipe Throughput	4071	4064	99%
Shell Scripts	4521	3969	87%
File Read	1051	1031	98%
File Write	1070	1066	99%

Table 3: Unixbench results comparison (p.2)

- Vanilla 2.6.18 kernel,
- OpenVZ 2.6.18-028stab025 kernel,
- OpenVZ 2.6.18-028stab025 kernel without pages sharing accounting.

Tests were run on Dual-Core Intel® Xeon™ CPU 3.20GHz machine with 3Gb of RAM.

Table 2 shows the results of comparing the vanilla kernel against the OpenVZ kernel without page-sharing accounting; and Table 3, for the vanilla kernel against the full beancounters patch.

5 Conclusions

Described in this article is the memory management made in "beancounters" subsystem. The userspace memory including RSS accounting and the kernel memory including network buffers management were described.

Beancounters architecture has proven its efficiency and flexibility. It has been used in OpenVZ kernels for all the time OpenVZ project exists.

The questions that are still unsolved are:

- TCP window management based on the beancounter resource availability;
- on-demand RSS accounting; and
- integration with the mainstream kernel.

References

[UB patch] Andrey Savochkin, *User Beancounter Patch*, `http://mirrors.paul.sladen.org/ftp.sw.com.sg/pub/Linux/people/saw/kernel/user_beancounter/UserBeancounter.html`

[RSS] Pavel Emelianov, *RSS fractions accounting* `http://wiki.openvz.org/RSS_fractions_accounting`

[AFTM] ATM Forum, *Traffic Management Specification, version 4.1*

[RLove] Robert Love, *Linux Kernel Development (2nd Edition)*, Ch. 15. Novell Press, January 12, 2005.

Manageable Virtual Appliances

David Lutterkort
Red Hat, Inc.
dlutter@redhat.com

Mark McLoughlin
Red Hat, Inc.
markmc@redhat.com

Abstract

Virtual Appliances are a relatively new phenomenon, and expectations vary widely on their use and precise benefits. This paper details some usage models, how requirements vary depending on the environment the appliance runs in, and the challenges connected to them. In particular, appliances for the enterprise have to fit into already existing infrastructure and work with existing sitewide services such as authentication, logging, and backup.

Appliances can be most simply distributed and deployed as binary images; this requires support in existing tools and a metadata format to describe the images. A second approach, basing appliances entirely on a metadata description that includes their setup and configuration, gives users the control they need to manage appliances in their environment.

1 Introduction

Virtual Appliances promise to provide a simple and cost-efficient way to distribute and deploy applications bundled with the underlying operating system as one unit. At its simplest, an appliance consists of disk images and a description how these disk images can be used to create one or more virtual machines that provide a particular function, such as a routing firewall, or a complete webmail system. Distributing a whole system instead of an application that the end user has to install themselves has clear benefits: installing the appliance is much simpler than setting up an application, the appliance supplier tests the appliance as a whole, and tunes all its components for the appliance's needs. In addition, appliances leave the appliance supplier much more latitude in the selection of the underlying operating system; this is particularly attractive for applications that are hard to port even amongst successive versions of the same distribution.

In addition to these installation-related benefits, some of the more general benefits of virtualization make appliances attractive, especially hardware isolation, better utilization of existing systems, and isolation of applications for improved security and reliability.

Because of their novelty, there is little agreement on what kind of appliance is most beneficial in which situation, how different end user requirements influence the tooling around appliances, and how users deploy and maintain their appliances. Examples of applications that are readily available as appliances range from firewalls and asterisk-based VoIP appliances, to complete Wikis and blogs, and database servers and LAMP stacks.

A survey of existing appliances makes one thing clear: there are many open questions around appliances; for them to fulfill their potential, these questions must be answered, and those answers must be backed by working, useful tools.

Comparing how current Linux distributions are built, delivered and managed to what is available for appliances points to many of these shortcomings: installing an appliance is a completely manual process, getting it to run often requires an intimate understanding of the underlying virtualization platform, and the ability to configure the virtual machine for the appliance manually; managing it depends too often on appliance-specific one-off solutions. As an example, for an appliance based on paravirtualized Xen, the host and the appliance must agree on whether to use PAE or not, a build-time configuration. A mismatch here leads to an obscure error message when the appliance is started. As another example, in an effort to minimize the size of the appliance image, appliance builders sometimes strip out vital information such as the package database, making it impossible to assess whether the appliance is affected by a particular security vulnerability, or, more importantly, update it when it is.

In addition, appliances bring some problems into the

spotlight that exist for bare-metal deployments, too, but are made more pressing because appliances rely on the distribution of entire systems. The most important of these problems is that of appliance configuration—very few appliances can reasonably be deployed without any customization by the end-user. While appliances will certainly be a factor in pushing the capabilities and use of current discovery and zeroconf mechanisms, it is unlikely that that is enough to address all the needs of users in the near future.

We restrict ourselves to appliances consisting of a single virtual machine. It is clear that there are uses for appliances made up of multiple virtual machines, for example, for three-tier web applications, with separate virtual machines for the presentation logic, the business logic, and the database tier. Such appliances though add considerable complexity to the single-VM case: the virtual machines of the appliance need to be able to discover each other, might want to create a private network for intra-appliance communication, and might want to constrain how the virtual machines of the appliance should be placed on hosts. For the sake of simplicity, and given that handling single-VM appliances well is a necessary stepping stone to multi-VM appliances, we exclude these issues and focus solely on single-VM appliances.

The rest of the paper is organized as follows: Section 2 discusses appliance builders and users, and how their expectations for appliances vary depending on the environment in which they operate, Section 3 introduces images and recipes, two different approaches to appliances, Sections 4 and 5 describe images and recipes in detail, and the final Section 6 discusses these two concepts, and some specific problems around appliances based on the example of a simple web appliance.

2 Builders and Users

Appliances available today cover a wide variety of applications and therefore target users: from consumer software to applications aimed at SMB's, to enterprise software. It is worthwhile to list in more detail the expectations and needs of different applications and target audiences—it is unlikely that the differences that exist today between a casual home user and a sysadmin in a large enterprise will disappear with appliances.

Two separate groups of people are involved with appliances: *appliance builders*, who create and distribute appliances, and *appliance users* who deploy and run them. We call the first group builders rather than developers for two reasons: while appliance creation will often involve a heavy dose of software development, it will, to a large extent, be very similar to traditional development for bare-metal systems and the need for appliance-specific development tools will be minimal; besides creating appliances through very specific development, they can also be created by assembling existing components with no or minimal custom development. The value of these appliances comes entirely from them being approved, pretested and tuned configurations of software. In that sense, a sysadmin in an enterprise who produces "golden images" is an appliance builder.

What all appliance builders have in common is that they need to go through repeated cycles of building the appliance from scratch, and running, testing and improving it. They therefore need ways to make these steps repeatable and automatable. Of course, they also share the goal of distributing the appliance, externally or internally, which means that they need to describe the appliance in as much detail as is necessary for users to deploy them easily and reliably.

Users and their expectations vary as much as the environments in which they operate, and expectations are closely coupled to the environment in which the appliance will run. Because of the differences in users, the requirements on the appliances themselves should naturally vary, too: whereas for consumer software ease of installation and configuration is crucial, appliances meant for an enterprise setting need to focus on being manageable in bulk and fitting into an environment with preexisting infrastructure and policies. As an example, a typical hardware appliance, a DSL router with a simple web configuration interface is a great consumer device, since it lets consumers perform complicated tasks with ease, while it is completely unsuited for a data center, where ongoing, automated management of a large number of routers is more important than the ease of first-time setup.

These differences have their repercussions for virtual appliances, too: an appliance developer for a consumer appliance will put more emphasis on an easy-to-use interface, whereas for enterprise appliances the emphasis will be on reliability, manageability and the problems that only appear in large-scale installations. Building the user interface for a consumer appliance is tightly coupled to the appliance's function, and we consider it part

of the custom development for the appliance; addressing the concerns of the enterprise user, the sysadmin, requires tools that go beyond the individual appliance, and therefore should be handled independently of the function of the appliance.

While for consumer appliances it is entirely reasonable to predefine what can and can not be configured about the appliance, this is not feasible for data center appliances: the infrastructures and policies at different sites are too varied to allow an appliance developer to anticipate all the changes that are required to make the appliance fit in. Often, these changes will have little bearing on the functioning of the appliance, and are concerned with issues such as logging to a central logging server, auditing shell access, monitoring the appliance's performance and resource usage etc. At the same time, enterprise users will demand that they can intervene in *any* aspect of the appliance's functioning should that become necessary, especially for security reasons.

Consumers and enterprise users also differ sharply in their expectations for a deployment tool: for a consumer, a simple graphical tool such as `virt-manager` is ideal, and adding features for downloading and running an appliance to it will provide consumers with a good basis for appliance use. For enterprise users, who will usually be sysadmins, it is important that appliances can be deployed, configured and managed in a way as automated as possible.

3 Images and Recipes

This paper details two different approaches to building and distributing appliances: *Appliance Images*, appliances distributed as one or more disk images and a description of how to create a virtual machine from them, and *Appliance Recipes*, appliances completely described in metadata.

Appliances can be built for any number of virtualization technologies and hypervisors; deployment tools for appliances should be able to understand appliance descriptions regardless of the virtualization technology they are built on, and should be able to at least let the user know ahead of time if a given appliance can be deployed on a given host. The appliance description therefore needs to carry information on the expected platform. Rather than encode this knowledge in appliance-specific deployment tools, we base the appliance description on libvirt and its XML metadata format for virtual machines, since libvirt can already deal with a number of virtualization platforms such as paravirtualized and fully-virtualized Xen hosts, qemu, and kvm and abstracts differences between them away. This approach also makes it possible for related tools such as `virt-manager` [1] and `virt-factory` [2] to integrate appliance deployment and management seamlessly.

The goal of both the image and recipe approach is to describe appliances in a way that integrates well with existing open-source tools, and to introduce as fewer additional requirements on the infrastructure as possible.

Deployment of Appliance Images is very simple, consisting of little more than creating a virtual machine using the disk images; for Appliance Recipes, additional steps amounting to a system install are needed. On the other hand, changing the configuration of Appliance Images is much harder and requires special care to ensure changes are preserved across updates, especially when updates are image-based, whereas Appliance Recipes provide much more flexibility in this area. In addition, Appliance Recipes provide a clear record of what exactly goes into the appliance, something that can be hard to determine with image-based appliances.

By virtue of consisting of virtual machines, appliances share some characteristics with bare-metal machines, and some of the techniques for managing bare-metal machines are also useful for appliances: for example, Stateless Linux [3] introduced the notion of running a system *readonly root* by making almost the entire file system readonly. For Stateless, this allows running multiple clients off the same image, since the image is guaranteed to be immutable. For image-based appliances, this can be beneficial since it cleanly delineates the parts of the appliance that are immutable content from those that contain user data. It does make it harder for the user to modify arbitrary parts of the appliance, especially its configuration, though Stateless provides mechanisms to make parts of the readonly image mutable, and to preserve client-specific persistent state.

Similarly, managing the configuration of machines is not a problem unique to appliances, and using the same tools for managing bare-metal and appliance configurations is very desirable, especially in large-scale enterprise deployments.

3.1 Relation to Virtual Machines

Eventually, an appliance, i.e., the artifacts given to the appliance user by the appliance builder, is used to create and run a virtual machine. If a user will only ever run the appliance in a single virtual machine, the appliance and the virtual machine are interchangeable, particularly for Appliance Images, in the sense that the original image the appliance user received is the one off which the virtual machine runs.

When a user runs several instances of the same appliance in multiple virtual machines, for example, to balance the load across several instances of the same web appliance, this relation is more complicated: appliances in general are free to modify any of their disk images, so that each virtual machine running the appliance must run off a complete copy of all of the appliance's disk images. The existence of the copies complicates image-based updates as the copy for each virtual machine must be found and updated, making it necessary to track the relation between original appliance image and the images run by each virtual machine. For Appliance Recipes, this is less of an issue, since the metadata takes the place of the original appliance image, and a separate image is created for every virtual machine running that appliance recipe.

3.2 Combining Images and Recipes

Appliance Images and Appliance Recipes represent two points in a spectrum of ways to distribute appliances. Image-based appliances are easy to deploy, whereas recipe-based appliances give the user a high degree of control over the appliance at the cost of a more demanding deployment.

A hybrid approach, where the appliance is completely described in metadata, and shipped as an image combines the advantages of both approaches: to enable this, the appliance builder creates images from the recipe metadata, and distributes both the metadata and the images together.

4 Appliance Images

The description of an Appliance Image, by its very nature, is focused on describing a virtual machine in a transportable manner. Such a description is not only useful for distributing appliances, but also anywhere where a virtual machine and all its parts need to be saved and recreated over long distances or long periods of time, for example for archiving an entire application and all its dependencies.

Appliance Images are also suitable for situations where the internals of the virtual machine are unimportant, or because the appliance is running an O/S that the rest of the environment can't understand in more detail.

With no internals exposed, the appliance then consists just of a number of virtual disks, descriptions of needed virtual hardware, most importantly a NIC, and constraints on which virtualization platform the appliance can run.

Appliance Images need to carry just enough metadata in their description to allow running them safely in an environment that knows nothing but the metadata about them. To enable distribution, Appliance Images need to be bundled in a standard format that makes it easy to download and install them. For simplicity, we bundle the metadata, as an XML file, and the disk images as normal tarballs.

Ultimately, we need to create a virtual machine based on the appliance metadata and the disk images; therefore, it has to be possible to generate a libvirt XML description of the virtual machine from the appliance's metadata. Using libvirt XML verbatim as the appliance description is not possible, since it contains some information, such as the MAC address for the appliance's NIC, that clearly should be determined when the appliance is deployed, not when it is built.

4.1 Metadata for Appliance Images

The metadata for an Appliance Image consists of three parts:

1. General information about the appliance, such as a name and human-readable label
2. The description of the virtual machine for the appliance
3. The storage for the appliance, as a list of image files

4.1.1 Virtual Machine Descriptor

The metadata included with an appliance needs to contain enough information to make a very simple decision that anybody (and any tool) wanting to run the appliance will be faced with: can this appliance run on this host? Since the differences between running a virtual machine on a paravirtualized and fully-virtualized host from an application's point of view are small, the appliance metadata allows describing them simultaneously with enough detail for tools to determine how to boot the appliance ahead of time; for each supported virtualization platform, the metadata lists how to boot the virtual machine on that platform, together with a description of the platform.

The boot descriptor roughly follows libvirt's `<os>` element: for fully-virtualized domains, it contains an indication of which bootloader and what boot device to use, and for paravirtualized domains, it either lists the kernel and initrd together or that `pygrub` should be used as well as the root device and possible kernel boot options. If the kernel/initrd are mentioned explicitly, they must be contained in the tarball used to distribute the appliance as separate files.

The platform description, based on libvirt's *capabilities*, indicates the type of hypervisor (for example, `xen` or `hvm`), the expected architecture (for example, `i686` or `ia64`), and additional features such as whether the guest needs `pae` or not.

Besides these two items, the virtual machine metadata lists how the disk images should be mapped into the virtual machine, how much memory and how many virtual CPUs it should receive, whether a (graphical) console is provided, and which network devices to create.

4.1.2 Disk Images

The disk images for the appliance are simple files; for the time being, we use only (sparse) raw files, though it would be desirable to use compressed `qcow` images for disks that are rarely written to.[1]

Disk images are classified into one of three categories to enable a simple update model where parts of the appliance are replaced on update. Such an update model requires that the appliance separates the user's data from the appliance's code, and keeps them on separate disks. The image categories are:

- *system* disks that contain the O/S and application and are assumed to not change materially over the appliance's lifetime
- *data* disks that contain application data that must be preserved across updates
- *scratch* disks that can be erased at will between runs of the appliance

The classification of disk images mirrors closely how a filesystem needs to be labeled for Stateless Linux: in a Stateless image, most files are readonly, and therefore belong on a system disk. Mutable files come in two flavors: files whose content needs to be preserved across reboots of the image, which therefore belong on a data disk, and files whose content is transient, and therefore belong on a scratch disk.

Updates of an Appliance Image can be performed in one of two ways: by updating from within, using the update mechanisms of the appliance's underlying operating system, for example, yum, or by updating from the outside, replacing some of the appliance's system disks. If the appliance is intended to be updated by replacing the system disks, it is very desirable, though not strictly necessary, that the system disks are mounted readonly by the appliance; to prepare the appliance for that, the same techniques as for Stateless Linux need to be applied, in particular, marking writable parts of the filesystem in `/etc/rwtab` and `/etc/statetab`.

Data disks don't have to be completely empty when the appliance is shipped: as an example, a web application with a database backend, bundled as an appliance will likely ship with a database that has been initialized and the application's schema loaded.

4.2 Building an Appliance Image

An appliance supplier creates the appliance by first installing and configuring the virtual machine for the appliance any way they see fit; typically this involves installing a new virtual machine, starting it, logging into it and making manual configuration changes. Even though

[1] Unfortunately, the Xen blktap driver has a bug that makes it impossible to use `qcow` images that were not created by Xen tools.

these steps will produce a distributable Appliance Image, they will generally not make building the Appliance Image reproducible in an automated manner. When this is important, for example, when software development rather than simple assembly is an important part of the appliance build process, the creation of the Appliance Image should be based on an Appliance Recipe, even if the appliance is ultimately only distributed as an image.

Once the appliance supplier is satisfied with the setup of the virtual machine, they create an XML description of the appliance, based on the virtual machine's characteristics. Finally, the virtual machine's disk is compressed and, together with the appliance descriptor, packed into a tarball.

This process contains some very mechanical steps, particularly creating the initial appliance descriptor and packing the tarball that will be supported by a tool. The tool will similarly support unpacking, installing, and image-based updating of the appliance.

4.3 Deploying an Appliance Image

An appliance user deploys an appliance in two steps: first, she downloads and installs the appliance into a well-known directory, uncompressing the disk images.

As a second step, the user simply runs `virt-install` to create the virtual machine for the appliance; `virt-install` is a command line tool generally used to provision operating systems into virtual machines. For Appliance Images, it generates the libvirt XML descriptor of the appliance's virtual machine from the appliance descriptor and additional user-provided details such as an explicit MAC address for the virtual machine's NIC, or whether and what kind of graphical console to enable.

To allow for multiple virtual machines running the same appliance concurrently, the disk images for the appliance are copied into per-VM locations, and `virt-install` records the relation between the appliance and the VM image. This is another reason why using `qcow` as the image format is preferrable to simple raw images, as it has a facility for *overlay images* that only record the changes made to a base image. With `qcow` images, instantiating a virtual machine could avoid copying the usually large system disks, creating only an overlay for them. Data disks, of course, still should be created through copying from the original appliance image, while scratch disks should be created as new files, as they are empty by definition.

It is planned to integrate Appliance Image deployment into `virt-manager`, too, to give users a simple graphical interface for appliance deployment.

In both cases, the tools check that the host requirements in the appliance descriptor are satisfied by the actual host on which the appliance is deployed.

4.4 Packaging Appliance Images as RPM's

Packaging appliances as tarballs provides a lowest-common-denominator for distributing appliances that is distribution, if not operating system, agnostic. Most distributions already have a package format tailored to distributing and installing software and tools built around them.

For RPM-based distributions, it seems natural to package appliances as RPMs, too. This immediately sets a standard for, amongst others, how appliances should be made available (as RPMs in yum repositories), how their authenticity can be guaranteed (by signing them with a key known to RPM), and how they are to be versioned.

5 Appliance Recipes

Appliance Recipes describe a complete virtual machine in metadata during its whole lifecycle from initial provisioning to ongoing maintenance while the appliance is in use. The recipe contains the specification of the appliance's virtual machine, which is very similar to that for Appliance Images, and a description of how the appliance is to be provisioned initially and how it is to be configured. In contrast to Appliance Images, it does not contain any disk images. An Appliance Recipe consists of the following parts:

1. An appliance descriptor describing the virtual machine; the descriptor is identical to that for Appliance Images, except that it must contain the size of each disk to be allocated for the virtual machine.

2. A kickstart file, used to initially provision the virtual machine from the recipe.

3. A puppet manifest, describing the appliance's configuration. The manifest is used both for initial provisioning, and during the lifetime of the appliance; updating the manifest provides a simple way to update existing appliances without having to reprovision them.

Before an appliance based on a recipe can be deployed in a virtual machine, the virtual machine's disks need to be created and populated, by installing a base system into empty disk images. Once the disk images have been created, deployment follows the same steps as that for an Appliance Image.

The Appliance Recipe shifts who builds the disk images from the appliance builder to the appliance user, with one very important difference: the appliance user has a complete description of the appliance, consumable by tools, that they can adapt to their needs. With that, the appliance user can easily add site-specific customizations to the appliance, by amending the appliance's metadata; for example, if all `syslog` messages are to be sent to a specific server, the appliance user can easily add their custom `syslog.conf` to the appliance's description. It also provides a simple mechanism for the appliance builder to leave certain parts of the appliance's configuration as deploy-time choices, by parameterizing and templating that part of the appliance's metadata.

We use kickstart, Fedora's automated installer, for the initial provisioning of the appliance image, and puppet, a configuration management tool, for the actual configuration of the appliance. It is desirable to expose as much of the appliance's setup to puppet and to only define a very minimal system with kickstart for several reasons: keeping the kickstart files generic makes it possible to share them across many appliances and rely only on a small set of well-tested stock of kickstart files; since kickstarting only happens when a virtual machine is first provisioned, any configuration the kickstart file performs is hard to track over the appliance's lifetime; and, most importantly, by keeping the bulk of the appliance's configuration in the puppet manifest it becomes possible for the appliance user to adapt as much as possible of the appliance to their site-specific needs. The base system for the appliance only needs to contain a minimal system with a DHCP client, `yum`, and the puppet client.

Strictly speaking, an Appliance Recipe shouldn't carry a full kickstart file, since it contains many directives that should be controlled by the appliance user, not the appliance builder, such as the timezone for the appliance's clock. The most important parts of the kickstart file that the appliance builder needs to influence are the layout of the appliance's storage and how disks are mounted into it, and the yum repositories needed during provisioning. The recipe should therefore only ship with a partial kickstart file that is combined with user- or site-specific information upon instantiation of the image.

5.1 Configuration Management

The notion of Appliance Recipes hinges on describing the configuration of a virtual machine in its entirety in a simple, human-readable format. That description has to be transported from the appliance builder to the appliance user, leaving the appliance user the option of overriding almost arbitrary aspects of the configuration. These overrides must be kept separate from the original appliance configuration to make clean upgrades of the latter possible: if the user directly modifies the original appliance configuration, updates will require cumbersome and error-prone merges.

The first requirement, describing the configuration of a system, is the bread and butter of a number of configuration management tools, such as cfengine, bcfg2, and puppet. These tools are also geared towards managing large numbers of machines, and provide convenient and concise ways to expressing the similarities and differences between the configuration of individual machines. We chose puppet as the tool for backing recipes, since it meets the other two requirements of making configurations transportable and overridable particularly well.

The actual setup and configuration of an appliance, i.e. the actual appliance functionality, is expressed as a *puppet manifest*. The manifest, written in puppet's declarative language, describes the configuration through a number of *resource definitions* that describe basic properties of the configuration, such as which packages have to be installed, what services need to be enabled, and what custom config files to deploy. Files can either be deployed as simple copies or by instantiating templates.

Resource definitions are grouped into *classes*, logical units describing a specific aspect of the configuration; for example, the definition of a class `webserver`

might express that the `httpd` package must be installed, the `httpd` service must be enabled and running, and that a configuration file `foo.conf` must be deployed to `/etc/httpd/conf.d`.

The complete configuration of an actual system is made up of mapping any number of classes to that system, a node in puppet's lingo. This two-level scheme of classes and nodes is at the heart of expressing the similarities and differences between systems, where, for example, all systems at a site will have their logging subsystem configured identically, but the setup of a certain web application may only apply to a single node.

Site-specific changes to the original appliance configuration fall into two broad categories: overriding core parts of the appliance's configuration, for example, to have its webserver use a site-specific SSL certificate, and adding to the appliance's setup, without affecting its core functionality, for example, to send all syslog messages to a central server.

These two categories are mirrored by two puppet features: overrides are performed by subclassing classes defined by the appliance and substituting site-specific bits in the otherwise unchanged appliance configuration. Additions are performed by mapping additional, site-specific classes to the node describing the appliance.

The configuration part of an Appliance Recipe consists of a puppet *module*, a self-contained unit made up of the classes and supporting files. Usually, puppet is used in client/server mode, where the configuration of an entire site is stored on a central server, the *puppetmaster*, and clients receive their configuration from it. In that mode, the appliance's module is copied onto the puppetmaster when the recipe is installed.

5.2 Deploying an Appliance Recipe

Deploying an Appliance Recipe requires some infrastructure that is not needed for Appliance Images, owing to the fact that the appliance's image needs to be provisioned first. Recipes are most easily deployed using `virt-factory`, which integrates all the necessary tools on a central server. It is possible though to use a recipe without `virt-factory`'s help; all that is needed is the basic infrastructure to perform kickstart-based installs with `virt-install`, and a puppetmaster from which the virtual machine will receive its configuration once it has been booted.

In preparation for the appliance instantiation, the appliance's puppet manifest has to be added to the puppetmaster by copying it to the appropriate place on the puppetmaster's filesystem. With that in place, the instantiation is entirely driven by `virt-install`, which performs the following steps:

1. Create empty image files for the virtual machine
2. Create and boot the virtual machine and install it according to the appliance's kickstart file
3. Bootstrap the puppet client during the install
4. Reboot the virtual machine
5. Upon reboot, the puppet client connects to the puppetmaster and performs the appliance-specific configuration and installation

5.3 Creating an Appliance Recipe

For the appliance builder, creating an Appliance Recipe is slightly more involved than creating an Appliance Image. The additional effort is caused by the need to capture the appliance's configuration in a puppet manifest. The manifest can simply be written by hand, being little more than formalized install instructions.

It is more convenient though to use cft [4], a tool that records changes made to a system's configuration and produces a puppet manifest from that. Besides noticing and recording changes made to files, it also records more complex and higher-level changes such as the installation of a package, the modification of a user, or the starting and enabling of a service.

With that, the basic workflow for creating an Appliance Recipe is similar to that for an Appliance Image:

1. Write (or reuse) a kickstart file and install a base system using `virt-install`
2. Start the virtual machine with the base system
3. Log into the virtual machine and start a cft session to capture configuration changes
4. Once finished configuring the virtual machine, finish the cft session and have it generate the puppet manifest

5. Generate the appliance descriptor for the virtual machine

6. Package appliance descriptor, kickstart file, and puppet manifest into an Appliance Recipe

Note that all that gets distributed for an Appliance Recipe is a number of text files, making them considerably smaller than Appliance Images. Of course, when the user instantiates the recipe, images are created and populated; but that does not require that the user downloads large amounts of data for this one appliance, rather, they are more likely to reuse already existing local mirrors of yum repositories.

6 Example

As an example, we built a simple web calendaring appliance based on kronolith, part of the horde PHP web application framework; horde supports various storage backends, and we decided to use PostgreSQL as the storage backend.

To separate the application from its data, we created a virtual machine with two block devices: one for the application, and one for the PostgreSQL data, both based on 5GB raw image files on the host. We then installed a base system with a DHCP client, `yum` and `puppet` on it and started the virtual machine.

After logging in on the console, we started a cft session and performed the basic setup of kronolith: installing necessary packages, opening port 80 on the firewall, and starting services such as `httpd` and `postgresql`. Following kronolith's setup instructions, we created a database user and schema for it, and gave the web application permission to connect to the local PostgreSQL database server. Using a web browser we used horde's administration UI to change a few configuration options, in particular to point it at its database, and to create a sample user.

These steps gave us both an Appliance Image and an Appliance Recipe: the Appliance Image consists of the two image files, the kernel and initrd for the virtual machine, and the appliance XML description. The Appliance Recipe consists of the appliance XML description, the kickstart file for the base system, and the puppet manifest generated by the cft session.

Even an application as simple as kronolith requires a small amount of custom development to be fully functional as an appliance. For kronolith, there were two specific problems that need to be addressed: firstly, kronolith sends email alerts of upcoming appointments to users, which means that it has to be able to connect to a mailserver, and secondly, database creation is not fully scripted.

We addressed the first problem, sending email, in two different ways for the image and the recipe variant of the appliance: for the image variant, kronolith's web interface can be used to point it to an existing mail hub. Since this modifies kronolith's configuration files in `/etc`, these files need to be moved to the data disk to ensure that they are preserved during an image-based update. For the recipe case, we turned the appropriate config file into a template, requiring that the user has to fill in the name of the mail hub in a puppet manifest before deployment.

The fact that database creation is not fully scripted is not a problem for the kronolith Appliance Image, since the database is created by the appliance builder, not the user. For the recipe case, recipe instantiation has to be fully automated; in this case though, the problem is easily addressed with a short shell script that is included with the puppet manifest and run when the appliance is instantiated.

Another issue illustrates how low-level details of the appliance are connected to the environment in which it is expected to run: as the appliance is used over time, it is possible that its data disk fills up. For consumer or SMB use, it would be enough to provide a page in the web UI that shows how full the appliance's disk is—though a nontechnical user will likely lack the skill to manually expand the data disk, and therefore needs the appliance to provide mechanisms to expand the data disk. The appliance can not do that alone though, since the file backing the data disk must be expanded from outside the appliance first. For this use, the appliance tools have to provide easy-to-use mechanisms for storage management.

For enterprise users, the issue presents itself completely differently: such users in general have the necessary skills to perform simple operations such as increasing storage for the appliance's data. But they are not very well served by an indication of how full the data disk is in the appliance's UI because such mechanisms scale

poorly to large numbers of machines; for the same reason, sending an email notification when the data disk becomes dangerously full is not all that useful either. What serves an enterprise user best is being able to install a monitoring agent inside the appliance. Since different sites use different monitoring systems, this is a highly site-dependent operation. With the recipe-based appliance, adding the monitoring agent and its configuration to the appliance is very simple, no different from how the same is done for other systems at the site. With the image-based appliance, whether this is possible at all depends on whether the appliance builder made it possible to gain shell access. Other factors, such as whether the appliance builder kept a package database in the appliance image and what operating system and version the appliance is based on, determine how hard it is to enable monitoring of the appliance.

Over time, and if appliances find enough widespread use, we will probably see solutions to these problems such as improved service discovery and standardized interfaces for monitoring and storage management that alleviate these issues. Whether they can cover all the use cases that require direct access to the insides of the appliance is doubtful, since in aggregate they amount to managing every aspect of a system from the outside. In any event, such solutions are neither mature nor widespread enough to make access and use of traditional management techniques unnecessary in the near future.

Details of the kronolith example, including the appliance descriptor and the commented recipe can be found on a separate website [5].

Acknowledgements

We wish to thank Daniel Berrange for many fruitful discussions, particularly around appliance descriptors, libvirt capabilities and the ins and outs of matching guests to hosts.

References

[1] `http://virt-manager.et.redhat.com/`

[2] `http://virt-factory.et.redhat.com/`

[3] `http://fedoraproject.org/wiki/StatelessLinux`

[4] `http://cft.et.redhat.com/`

[5] `http://people.redhat.com/dlutter/kronolith-appliance.html`

Everything is a virtual filesystem: libferris

Ben Martin
affiliation pending
monkeyiq@users.sf.net

Abstract

Libferris is a user address space virtual (Semantic) filesystem. Over the years it has grown to be able to mount relational databases, XML, and many applications including X Window. Rich support for index and search has been added. Recently the similarities between modern Linux kernel filesystem, Semantic Filesystems and the XML data model has been exploited in libferris. This leads to enabling XSLT filesystems and the evaluation of XQuery directly on a filesystem. XQueries are evaluated using both indexing and shortcut loading to allow things like db4 files or kernel filesystems with directory name caching to be used in XQuery evaluation so that large lookup tables can be efficiently queried. As the result of XQuery is usually XML—As the similarities between XML and filesystems are discussed, the option is there for queries to generate filesystems.

Prior knowledge of the existance of Extended Attributes as well as some familiarity with XML and XQuery will be of help to the reader.

1 Introduction

Libferris [2, 9, 11, 13, 14] is a user address space virtual filesystem [1]. The most similar projects to libferris are gnome-vfs and kio_slaves. However, the scope of libferris is extended both in terms of its capability to mount things, its indexing and its metadata handling.

Among its "non conventional" data sources, libferris is able to mount relational databases, XML, db4, Evolution, Emacs, Firefox and X Window.

The data model of libferris includes rich support for unifying metadata from many sources and presenting applicable metadata on a per filesystem object basis. Indexing and querying based on both fulltext and metadata predicates complements this data model allowing users to create virtual filesystems which satisfy their information need. It should be noted that metadata can be associated with any filesystem object, for example a tuple in a mounted database.

The paper now moves to discuss what semantic filesystems are and in particular what libferris is, and how it relates to the initial designs of a semantic filesystems. The similarities and differences between the libferris, traditional Linux kernel filesystem and XML data models is then discussed with mention of how issues with data model differences have been resolved where they arise. The focus is then turned to information indexing and search. The indexing section is more an overview of previous publications in the area to give the reader familiarity for the example in the XQuery section. The treatment of XQuery evaluation both directly on a filesystem and on its index then rounds out the paper.

2 Semantic Filesystems

The notion of a semantic filesystem was originally published by David K. Gifford et al. in 1991 [4].

A semantic file system differs from a traditional file system in two major ways:

- Interesting meta data from files is made available as key-value pairs.
- Allowing the results of a query to be presented as a virtual file system.

The notion of interesting meta data is similar to modern Linux kernel Extended Attributes. The main difference being that meta data in a Semantic Filesystem is inferred at run time whereas Linux kernel Extended Attributes are read/write persisted byte streams. In the context of libferris, the term Extended Attributes can refer to both persisted and inferred data. In this way the Extended

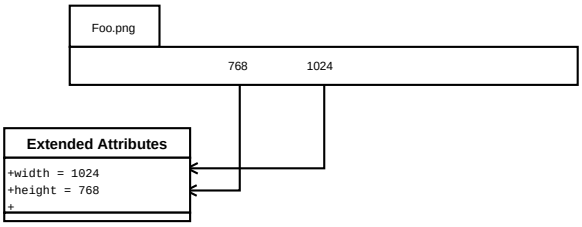


Figure 1: The image file Foo.png is shown with it's byte contents displayed from offset zero on the left extending to the right. The png image transducer knows how to find the metadata about the image file's width and height and when called on will extract or infer this information and return it through a metadata interface as an Extended Attribute.

Attributes in libferris have been virtualized along with the filesystem itself. The term Extended Attributes will be used in the libferris sense unless otherwise qualified.

In a Semantic filesystem interesting meta data is extracted from a file's byte content using what are referred to as transducers [4]. An example of a transducer would be a little bit of code that can extract the width of a specific image file format. A transducer to extract some image related metadata is shown conceptually in Fig.1. Image dimensions are normally available at specific offsets in the file's data depending on the image format. A transducer which understands the PNG image encoding will know how to find the location of the width and height information given a PNG image file.

Queries are submitted to the file system embedded in the path and the results of the query form the content of the virtual directory. For example, to find all documents that have been modified in the last week one might read the directory "/query/(mtime>=begin last week)/". The results of a query directory are calculated every time it is read. Any metadata which can be handled by the transducers [4] (metadata extraction process) can be used to form the query.

Libferris allows the filesystem itself to automatically chain together implementations. The filesystem implementation can be varied at any file or directory in the filesystem. For example, in Figure 2 because an XML file has a hierarchical structure it might also be seen as a filesystem. The ability to select a different implementation at any directory in a URL requires various filesystems to be overlaid on top of each other in order to present a uniform filesystem interface.

When the filesystem implementation is varied at a file or directory then two different filesystem handlers are active at once for that point. The left side of Figure 2 is shown with more details in Figure 3. In this case both the operating system kernel implementation and the XML stream filesystem implementation are active at the URL `file://tmp/order.xml`. The XML stream implementation relies on the kernel implementation to provide access to a byte stream which is the XML file's contents. The XML implementation knows how to interpret this byte stream and how to allow interaction with the XML structure though a filesystem interface.

Note that because in the above the XML implementation can interact with the operating system kernel implementation to complete its task this is subtly different to standard UNIX mounting where a filesystem completely overrides the mount point.

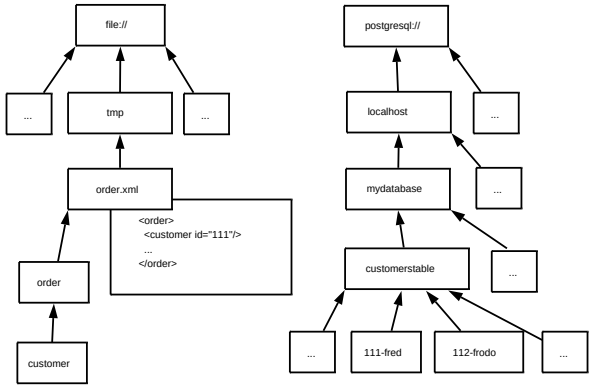


Figure 2: A partial view of a libferris filesystem. Arrows point from children to their parents, file names are shown inside each rectangle. Extended Attributes are not shown in the diagram. The box partially overlapped by `order.xml` is the contents of that file. On the left side, an XML file at path /tmp/order.xml has a filesystem overlaid to allow the hierarchical data inside the XML file to be seen as a virtual filesystem. On the right: Relational data can be accessed as one of the many data sources available though libferris.

The core abstractions in libferris can be seen as the ability to offer many filesystem implementations and select from among them automatically where appropriate for the user, the presentation of key-value attributes that files posses, a generic stream interface [6] for file and metadata content, indexing services and the creation of arbitrary new files.

Filesystem implementations allow one to drill into com-

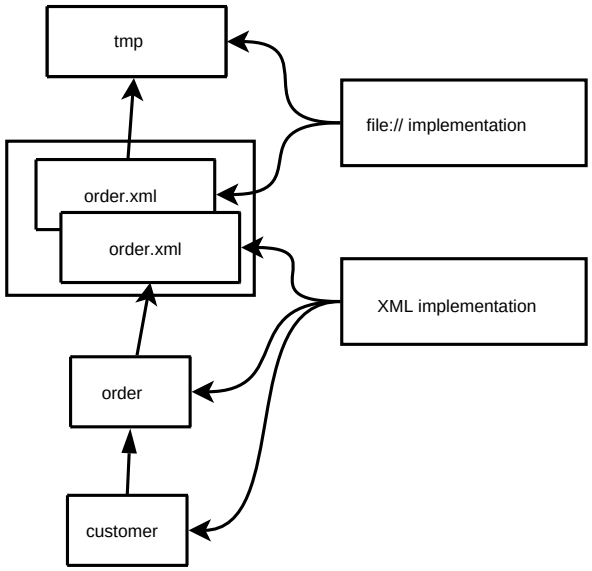


Figure 3: The filesystem implementation for an XML file is selected to allow the hierarchical structure of the XML to be exposed as a filesystem. Two different implementations exist at the "order.xml" file level: an implementation using the operating system's kernel IO interface and an implementation which knows how to present a stream of XML data as a filesystem. The XML implementation relies on the kernel IO implementation to provide the XML data itself.

posite files such as XML, ISAM[1] databases or tar files and view them as a file system. This is represented in Figure 2. Having the virtual filesystem able to select among filesystem implementations in this fashion allows libferris to provide a single file system model on top of a number of heterogeneous data sources.[2]

Presentation of key-value attributes is performed by either storing attributes on disk or by creating synthetic attributes who's values can be dynamically generated and can perform actions when their values are changed. Both stored and generated attributes in libferris are referred to simply as Extended Attributes (EAs). Examples of EAs that can be generated include the width and height of an image, the bit rate of an mp3 file or the MD5[3] hash of a file. This arrangement is shown in Figure 4.

For an example of a synthetic attribute that is writable consider an image file which has the key-value EA `width=800` attached to it. When one writes a value of `640` to the EA `width` for this image then the file's image data is scaled to be only 640 pixels wide. Having performed the scaling of image data the next time the `width` EA is read for this image it will generate the value `640` because the image data is `640` pixels wide. In this way the differences between generated and stored attributes are abstracted from applications.

Another way libferris extends the EA interface is by offering schema data for attributes. Such meta data allows for default sorting orders to be set for a datatype, filtering to use the correct comparison operator (integer vs. string comparison), and GUIs to present data in a format which the user will find intuitive.

3 Data models: XML and Semantic Filesystems

As the semantic filesystem is designed as an extension of the traditional Unix filesystem data model the two are very similar. Considering the relatively new adoption of Extended Attributes to kernel filesystems the data models between the two filesystem types are identical.

The main difference is the performance differences between deriving attributes (semantic filesystem and transducers) or storing attributes (Linux kernel Extended Attributes). Libferris extends the more traditional data model by allowing type information to be specified for its Extended Attributes and allowing many binary attributes to form part of a classification ontology [12, 7].

Type information is exposed using the same EA interface. For example an attribute `foo` would have `schema:foo` which contains the URL of the schema for the `foo` attribute. To avoid infinite nesting the schema for `schema:foo`, ie, `schema:schema:foo` will always have the type schema and there will be no `schema:schema:schema:foo`.

As the libferris data model is a superset of the standard Linux kernel filesystem data model one may ignore libferris specific features and consider the two data models in the same light. This is in fact what takes place when libferris is exposed as a FUSE filesystem [1].

[1] Indexed Sequential Access Method, e.g., B-Tree data stores such as Berkeley db.

[2] Some of the data sources that libferris currently handles include: http, ftp, db4, dvd, edb, eet, ssh, tar, gdbm, sysV shared memory, LDAP, mbox, sockets, mysql, tdb, and XML.

[3] MD5 hash function RFC, `http://www.ietf.org/rfc/rfc1321.txt`

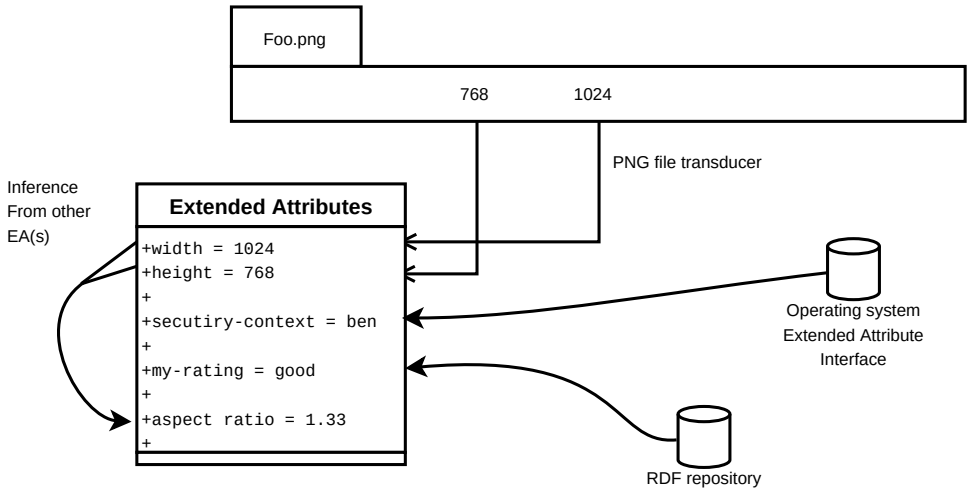


Figure 4: Metadata is presented via the same Extended Attribute (EA) interface. The values presented can be derived from the file itself, derived from the values of other EA, taken from the operating system's Extended Attribute interface or from an external RDF repository.

Taking an abstract view of the data model of libferris one arrives at: files with byte contents, files nested in a hierarchy and arbitrary attributes associated with each file. This is in many ways similar to the data model of XML: elements with an ordered list of byte content, elements nested in an explicit ordering with attributes possibly attached to each element.

The differences between the data models may raise issues and require explicit handling. The differences have been found to be:

- XML elements can contain multiple contiguous bytes serving as their "contents." Files may have many sections of contiguous bytes separated by holes. Holes serve to allow the many sections of contiguous bytes to appear in a single offset range. For example, I could have a file with the two worlds "alice" and "wonderland" logically separated by 10 bytes. The divergence of the data models in this respect is that the many sections of contiguous bytes in an XML element are not explicitly mapped into a single logical contiguous byte range.

- XML elements are explicitly ordered by their physical location in the document. For any two elements with a common parent element it will be apparent which element comes "before" the other. Normally files in a filesystem are ordered by an implementation detail—their inode. The inode is a unique number (across the filesystem itself) identifying that file. Many tools which interact with a filesystem will sort a directory by the file name to be more palatable to a human reader.

- The notions of file name and element name have different syntax requirements. A file name can contain any character apart from the "/" character. There are much more stringent requirements on XML element names—no leading numbers, a large range of characters which are forbidden.

- For all XML elements with a common parent it is not required that each child's name be unique. Any two files in a directory *must* have different names.

The differences are shown in Figure 5.

The identification of this link between data models and various means to address the issues where differences arise helps both the semantic filesystem and XML communities by bringing new possibilities to both. For example, the direct evaluation of XQuery on a semantic filesystem instead of on an XML document. The blurring of the filesystem and XML also allows modern Office suites to directly manipulate filesystems [14].

The file name uniqueness issue is only present if XML is being seen as a semantic filesystem. In this case it can be acceptable to modify the file name to include a unique number as a postfix. In cases such as resolution of XPath

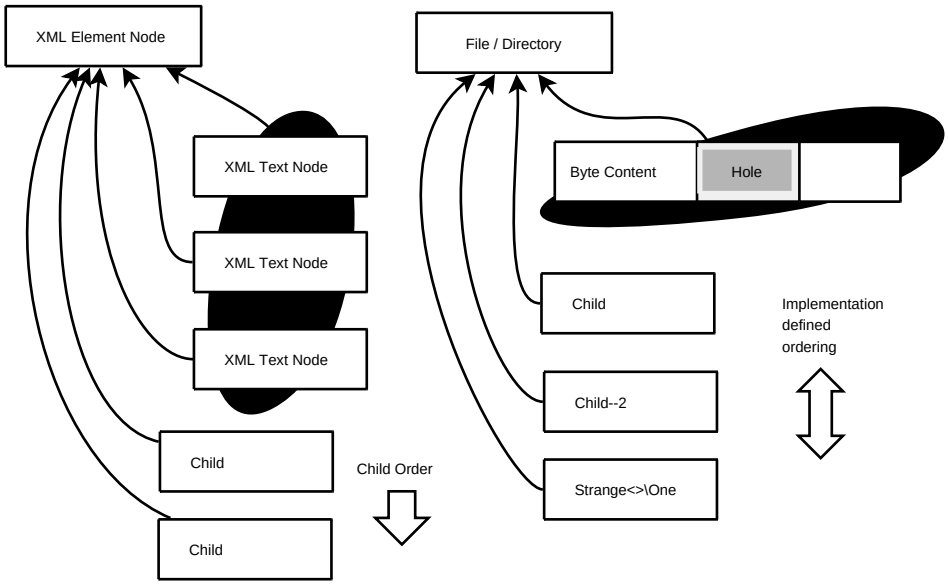


Figure 5: On the left an XML Element node is shown with some child nodes. On the right a filesystem node is shown with some similar child nodes. Note that XML Text nodes can be considered to provide the byte content of the synonymous filesystem abstraction but metadata about their arrangement can not be easily communicated. Child nodes in the XML side do not need to have unique names for the given parent node and maintain a strict document order. Child nodes on the filesystem side can contain more characters in their file names but the ordering is implementation defined by default.

or XQueries file names should be tested without consideration of the unique postfix so that query semantics are preserved.

As XML elements can not contain the "/" character exposing XML as a semantic filesystem poses no issue with mapping XML element names into file names. Unfortunately the heavy restrictions on XML element name syntax does present an issue. The most convenient solution has been found to be mapping illegal characters in file names into a more verbose description of the character itself. For example a file name "foo<bar>.txt" might be canonicalized to an XML element name of "foo-lt-bar-gt.txt". The original unmodified file name can be accessed through a name spaced XML attribute on the XML element.

XML element ordering can be handled by exposing the place that the XML element appeared in document order. For example, a document with "b" containing "c,d,e" in that order the "c" file would have a place of zero, and "e" would be two. With this attribute available the original document ordering can be obtained through the semantic filesystem by sorting the directory on that attribute. As there is no (useful) document ordering for a filesystem this is not an issue when exposing a filesystem as XML.

There is no simple solution to the fact that XML elements can have multiple text nodes as children. In cases where XML with multiple child nodes exist they are merged into a single text node containing the concatenation in document order of the child text nodes. Files with holes are presented as though the hole contained null bytes.

4 Information Search

In recent years much emphasis has been placed on so called "Desktop search". Few machines exist as islands and as such the index and search capabilities of any non trivial system must allow seamless integration of Internet, Intranet and desktop sources. The term "filesystem search" at least removes the connotations that search is limited to the desktop alone.

Details of indexing have been presented in prior publications [10, 9, 11]. Briefly the indexing capabilities in libferris are exposed through plugins. Much of the emphasis has been placed on indexing of metadata leaving full

text indexing [17] to implementations such as Lucene and TSearch2. Two of the main metadata plugins use sorted Berkeley db4 files or a PostgreSQL database.

The chosen query syntax is designed based on the "The String Representation of LDAP Search Filters" [5]. This is a very simple syntax which provides a small set of comparative operators to make `key operator value` terms and a means to combine these terms with boolean `and`, `or` and `not`. All terms are contained in parenthesis with boolean combining operators located before the terms they operate on.

The comparative operators have been enhanced and in some cases modified from the original semantics [5]. Syntax changes include the use of == instead of = for equality testing. Approximate matching ~= was dropped in favor of regular expression matching using perl operator syntax =~. Operators which are specific to the LDAP data model have been removed. The operators and semantics are presented in Table 1. Coercion of `rvalue` is performed both for sizes and relative times. For example, "begin today" will be converted into the operating system's time representation for the start of the day that the query is executed.

OP	Semantics
=~	`lvalue` matches the regular expression in `rvalue`
==	`lvalue` is exactly equal to `rvalue`
<=	`lvalue` is less than or equal to `rvalue`
>=	`lvalue` is greater than or equal to `rvalue`

Table 1: Comparative operators supported by the libferris search syntax. The operators are used infix, there is a key on the left side and a value on the right. The key is used to determine which EA is being searched for. The `lvalue` is the name of the EA being queried. The `rvalue` is the value the user supplied in the query.

Resolution of the `and` and `or` is performed (conceptually) by merging the sets of matching file names using either an intersection or union operation respectively. The semantics of negation are like a set subtraction: the files matching the negated subquery are removed from the set of files matching the portion of the query that the negation will combine with. If negation is applied as the top level operation then the set of files to combine with is considered to be the set of all files. The nesting of `and`, `or` and `not` will define what files the negation subquery will combine with. As an example of negation resolution consider the `fquery` which combines a width search with a negated size search: `(&(width<=640)(!(size<=900)))`. The set of files which have a width satisfying the first subquery are found and we call this set A. The set of files which have a size matching the second part of the query, ie, `size<=900` are found and we call this set B. The result of the query is $A \setminus B$.

The eaq:// virtual filesystem takes a query as a directory name and will populate the virtual directory with files matching the query. Other closely related query filesystems are the eaquery:// tree. The eaquery:// filesystem is has slightly longer URLs but it allows you to set limits on the number of results returned and to set how conflicting file names are resolved. Some example queries are shown in Figure 6. Normally a file's URL is used as its file name for eaquery:// filesystems. The shortnames option uses just the file's name and when two results from different directories happen to have the exact same file name it appends a unique number to one of the result's file names. This is likely to happen for common file names such as README.

Full text queries can be evaluated using the `fulltextquery://` or `ftxq://` URL schemes. Both metadata and fulltext predicates can be evaluated to produce a single result filesystem [11].

One major area where the index and search in libferris diverges from similar tools is the application of Formal Concept Analysis (FCA) [3]. FCA can be seen as unsupervised machine learning and is a formal method for dealing with the natural groupings within a given set of data. The result of FCA is a Concept Lattice. A Concept Lattice has many formal mathematical properties but may be considered informally as a specialization hierarchy where the further down a lattice one goes the more attributes the files in each node have. Files can be in multiple nodes at the same time. For example, if there are two attributes (mtime>=begin last week) and (mtime>=begin last month) then a file with the first attribute will also have the latter.

Using the SELinux type and identity of the example 201,759 files the concept lattice shown in Figure 7 is generated. The concept 11 in the middle of the bottom row shows that user_u identity is only active for 3 fonts_t typed files. Many of the links to the lower con-

cepts are caused by the root and system identities being mutually exclusive while the system identity combines with every attribute that the root identity does.

Readers interested in FCA with libferris should see [8, 16, 15].

4.1 XQuery

Being able to view an entire filesystem as an XML data model allows the evaluation of XQuery directly on the filesystem.

Libferris implements the native XQilla data model and attempts to offer optimizations to XQuery evaluation. Some of the possibilities of this are very nice, bringing together the db4, Postgresql, XML and file:// in a single XQuery.

There are also many efficiency gains that are available by having multiple data sources (filesystems) available. In an XML only query if you are looking up a user by a key things can be very slow. If you copy the users info into a db4 file and use libferris to evaluate your XQuery then a user lookup becomes only a handful of db4 btree or hash lookups.

Mounted postgresql functions allow efficient access to relational data from XQuery. Postgresql function arguments are passed through the file path from XQuery variables. This is a reasonable stop gap to being able to use prepared SQL statements with arguments as a filesystem. If the result is only a hand full of tuples with it will be very quick for libferris to make available to the xquery as a native document model.

A postgresql function is setup as shown in Figure 8. This can subsequently be used as any other read only filesystem as shown in Figure 9. A simple XQuery is shown in Figure 10 and its evaluation in Figure 11.

A major area which is currently optimized in the evaluation of XQuery with libferris is the evaluation of XPath expressions. This is done in the `Node::getAxisResult()` method specifically when the axis is `XQStep::CHILD`. Both files and directories are represented as Context objects in libferris. When seeking a specific child node `Context::isSubContextBound()` is used to quickly test if such a child exists. The `isSubContextBound()` method indirectly calls `Context::priv_getSubContext()` which is where filesystem plugins can offer the ability to read a directory piecewise.

The normal opendir(3), readdir(3) sequence of events to read a directory can be preempted by calls to `Context::priv_getSubContext()` to discover partial directory contents. As libferris is a virtual filesystem some other filesystem implementations also implement `Context::priv_getSubContext()` and piecewise directory reading. A specific example is the db4 filesystem which allows very efficient loading of a hand full of files from a large directory.

Where the direct evaluation on a filesystem as shown above becomes too slow the filesystem indexes can also be used in an XQuery by making an XPath that uses the filesystem interface to queries shown in Fig. 6.

A more complete example is shown in Figure 12. This uses the filesystem index and search along with XQuery variables to search for files which contain a person in a given location as a boolean `AND` style full text query. Note that multiple use of indexes, in particular the use of federated filesystem indexes [11] is possible together with immediate evaluation of other queries on db4 or RDF files to generate a combined result.

5 The Future

Closer integration of XML and libferris and in particular the ability to arbitrarily stack the two in any order. For example, being able to run an XQuery and take its results as the input to `xsltfs://` to generate an office document to edit with Open Office. As `xsltfs://` does not enforce a strict isomorphism between filesystems the resulting document when edited and saved could effect changes on both the underlying filesystem objects that the XQuery dealt with as well as any other desired side effects.

More efficient and user friendly access to the Formal Concept Analysis in libferris. There are still some complex persistence and processing tasks which need to be improved before the use of FCA on filesystems will see broad adoption.

References

[1] Fuse, http://fuse.sf.net. Visited Fed 2007.

```
# All files modified recently
$ ferrisls -lh "eaq://(mtime>=begin last week)"

# Same as above but limited to 100 results
# as an XML file
$ ferrisls --xml \
"eaquery://filter-100/(mtime>=begin last week)"

# limit of 10,
# resolve conflicts with version numbers
# include the desired metadata in the XML result
$ ferrisls --xml \
  --show-ea=mtime-display,url,size-human-readable \
"eaquery://filter-shortnames-10/(mtime>=blast week)"
```

Figure 6: Query results as a filesystem.

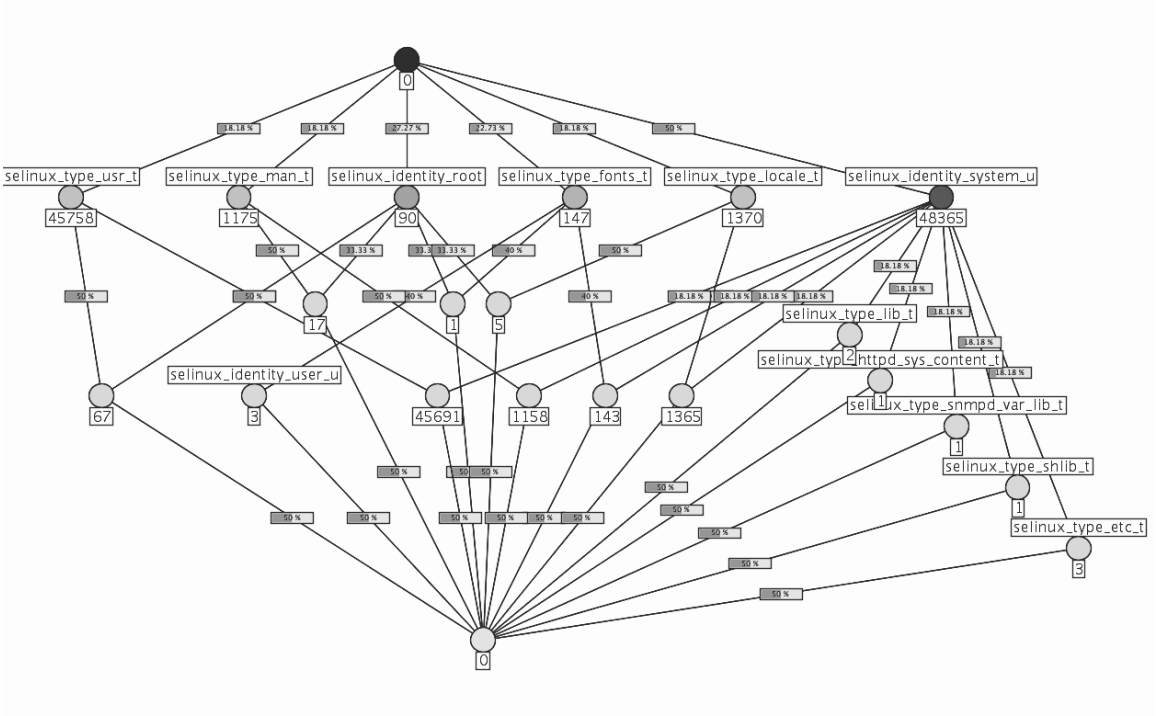


Figure 7: Concept lattice for SELinux type and identity of files in /usr/share/ on a Fedora Core 4 Linux installation. The Hasse diagram is arranged with three major sections; direct parents of the root are in a row across the top, refinements of selinux_identity_system_u are down the right side with combinations of the top row in the middle and left of the diagram. Attribute are shown above a node and they apply transitively to all nodes reachable downwards. The number of files in each node is shown below it.

```
$ psql junkdb
# CREATE TYPE junk_result
    AS (f1 int, f2 text);
# drop function junk( int, int );
# CREATE FUNCTION junk( int, int )
   returns setof junk_result
AS
$BODY$
DECLARE
  iter int;
rec junk_result;
BEGIN
  iter = $1;
for rec in select $1*10,$2*100 union
   select $1 * 100, $2 * 1000
LOOP
        return next rec;
END LOOP;
return;
END;
$BODY$
  LANGUAGE 'plpgsql' ;
# exit
```

Figure 8: Setting up a PostgreSQL function to be mounted by libferris

```
$ ferrisls --show-ea=f1,f2,name  --xml
"postgresql://localhost/play/junk(1,2)"
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<ferrisls>

  <ferrisls f1="" f2="" name="junk(1,2)"
url="postgresql:///localhost/play/junk(1,2)">
    <context f1="10" f2="200" name="10-200"/>
    <context f1="100" f2="2000" name="100-2000"/>
  </ferrisls>

</ferrisls>

$ fcat postgresql://localhost/play/junk(1,2)/10-200
...

$ ferriscp \
    postgresql://localhost/play/junk(1,2)/10-200 \
    /tmp/plan8.xml
```

Figure 9: Viewing the output of a PostgreSQL function with ferrisls

```
$ cat fdoc-pg.xq
<data>
 {
  for $b in ferris-doc("postgresql://localhost/play/junk(1,2)")
  return $b
 }
</data>
```

Figure 10: A Trivial XQuery to show the output of calling a PostgreSQL function through libferris

```
$ ferris-xqilla --show-ea=f1,f2,name fdoc-pg.xq
<?xml version="1.0"?>
<data>
  <junk_oper_1_comma_2_cper_ name="junk(1,2)">10,200
<number_10_dash_200 f1="10" f2="200" name="10-200">
&lt;context  f1="10"  f2="200"  /&gt;
</number_10_dash_200>
<number_100_dash_2000 f1="100" f2="2000" name="100-2000">
&lt;context  f1="100"  f2="2000"  /&gt;
</number_100_dash_2000></junk_oper_1_comma_2_cper_>
</data>
```

Figure 11: The evaluation of the XQuery in Figure 10. The embedded `<` etc. shown below come from the "content" of the file which in this case is the same as the above ferrisls command.

```
$ cat xquery-index.xq
declare variable $qtype    := "boolean";
declare variable $person   := "alice";
declare variable $location := "wonderland";
<data>
 {
  for $idx in ferris-doc( concat("fulltextquery://", $qtype, "/",
           $person, " ", $location))
    for $res in $idx/*
       return
    <match
            name="{ $res/@name }" url="{ $res/@url }"
            modification-time="{ $res/@mtime-display }"
           >
    </match>
 }
</data>

$ ferris-xqilla xquery-index.xq
<?xml version="1.0"?>
<data>
  <match modification-time="99 Jul 27 12:53"
     name="file:///.../doc/CommandLine/command.txt ...>
  <match modification-time="00 Mar 11 06:58"
     name="file:///.../doc/Gimp/Grokking-the-GIMP-v1.0/node8.html
     ...>
...</data>
```

Figure 12: Running an XQuery which uses filesystem index and search. The `idx` XQuery variable will be the virtual directory containing the query results and for the sake of clarity the `idx` is then looped over explicitly in the XQuery. The person and location can easily be obtained from other sources making the fulltext query portion complement a larger information goal.

[2] libferris, http://witme.sf.net/libferris.web/. Visited Nov 2005.

[3] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer–Verlag, Berlin Heidelberg, 1999.

[4] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. Jr O'Toole. Semantic file systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, ACM SIGOPS, pages 16–25, 1991.

[5] Network Working Group. Rfc 2254 - the string representation of ldap search filters, http://www.faqs.org/rfcs/rfc2254.html. Visited Sep 2003.

[6] Angelike Langer and Klaus Kreft. *Standard C++ IOStreams and Locales: Advanced programmer's Guide and Reference*. Addison Wesley, Reading, Massachusetts 01867, 2000.

[7] Ben Martin. File system wide file classification with agents. In *Australian Document Computing Symposium (ADCS03)*. University of Queensland, 2003.

[8] Ben Martin. Formal concept analysis and semantic file systems. In Peter W. Eklund, editor, *Concept Lattices, Second International Conference on Formal Concept Analysis, ICFCA 2004, Sydney, Australia, Proceedings*, volume 2961 of *Lecture Notes in Computer Science*, pages 88–95. Springer, 2004.

[9] Ben Martin. Filesystem indexing with libferris. *Linux Journal*, 2005(130):7, 2005.

[10] Ben Martin. A virtual filesystem on steroids: Mount anything, index and search it. In *Proceedings of the 12th International Linux System Technology Conference (Linux-Kongress 2005)*. GUUG e.V. / Lehmanns / Ralf Spenneberg, 2005.

[11] Ben Martin. Federated desktop and file server search with libferris. *Linux Journal*, 2006(152):8, 2006.

[12] Ben Martin. Geotagging files with libferris and google earth (linux.com), April 2006.

[13] Ben Martin. The world is a libferris filesystem. *Linux Journal*, 2006(146):7, 2006.

[14] Ben Martin. Virtual filesystems are virtual office documents. *Linux Journal*, 2007(154):8, 2007.

[15] Ben Martin and Peter W. Eklund. Spatial indexing for scalability in fca. In Rokia Missaoui and Jürg Schmid, editors, *ICFCA*, volume 3874 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2006.

[16] Ben Martin and Peter W. Eklund. Custom asymmetric page split generalized index search trees and formal concept analysis. In *ICFCA*, 2007.

[17] Ian H. Witten, Alistar Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 340 Pine Street, San Francisco, CA 94104-3205, USA, 1999.