Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation

Thomas E. Hart[1]*, Paul E. McKenney[2], and Angela Demke Brown[1]

[1]University of Toronto
Dept. of Computer Science
Toronto, ON M5S 2E4 CAN
{tomhart, demke}@cs.toronto.edu

[2]IBM Beaverton
Linux Technology Center
Beaverton, OR 97006 USA
paulmck@us.ibm.com

Abstract

Achieving high performance for concurrent applications on modern multiprocessors remains challenging. Many programmers avoid locking to improve performance, while others replace locks with non-blocking synchronization to protect against deadlock, priority inversion, and convoying. In both cases, dynamic data structures that avoid locking, require a memory reclamation scheme *that reclaims nodes once they are no longer in use.*

The performance of existing memory reclamation schemes has not been thoroughly evaluated. We conduct the first fair and comprehensive comparison of three recent schemes—quiescent-state-based reclamation*,* epoch-based reclamation*, and* hazard-pointer-based reclamation*—using a flexible microbenchmark. Our results show that there is no globally optimal scheme. When evaluating lockless synchronization, programmers and algorithm designers should thus carefully consider the data structure, the workload, and the execution environment, each of which can dramatically affect memory reclamation performance.*

1 Introduction

As multiprocessors become mainstream, multithreaded applications will become more common, increasing the need for efficient coordination of concurrent accesses to shared data structures. Traditional locking requires expensive atomic operations, such as compare-and-swap (CAS), even when locks are uncontended. For example, acquiring and releasing an uncontended spinlock requires over 400 cycles on an IBM® POWER™ CPU. Therefore, many researchers recommend avoiding locking [2, 7, 22]. Some systems, such as Linux™, use *concurrently-readable* synchronization, which uses locks for updates but not for reads.

*Supported by an NSERC Canada Graduate Scholarship.

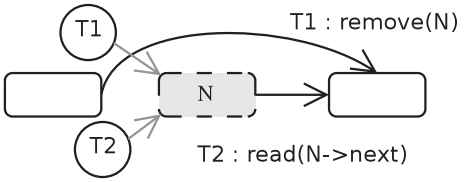

Figure 1. Read/reclaim race.

Locking is also susceptible to priority inversion, convoying, deadlock, and blocking due to thread failure [3, 10], leading researchers to pursue *non-blocking* (or *lock-free*) synchronization [6, 12, 13, 14, 16, 29]. In some cases, lock-free approaches can bring performance benefits [25]. For clarity, we describe all strategies that avoid locks as *lockless*.

A major challenge for lockless synchronization is handling the *read/reclaim races* that arise in dynamic data structures. Figure 1 illustrates the problem: thread $T1$ removes node N from a list while thread $T2$ is referencing it. N's memory must be reclaimed to allow reuse, lest memory exhaustion block all threads, but such reuse is unsafe while $T2$ continues referencing N. For languages like C, where memory must be explicitly reclaimed (e.g. via `free()`), programmers must combine a *memory reclamation scheme* with their lockless data structures to resolve these races.[1] Several such reclamation schemes have been proposed.

Programmers need to understand the semantics and the performance implications of each scheme, since the overhead of inefficient reclamation can be worse than that of locking. For example, *reference counting* [5, 29] has high overhead in the base case and scales poorly with data-structure size. This is unacceptable when performance is the motivation for lockless synchronization. Unfortunately, there is no single optimal scheme and existing work is relatively silent on factors affecting reclamation performance.

[1]Reclamation is subsumed into automatic garbage collectors in environments that provide them, such as Java™.

1-4244-0054-6/06/$20.00

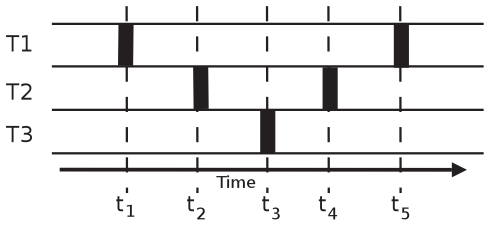

Figure 2. Illustration of QSBR. Black boxes represent quiescent states.

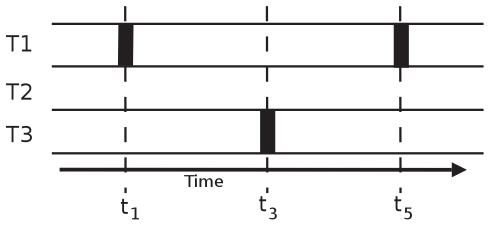

Figure 3. QSBR is inherently blocking.

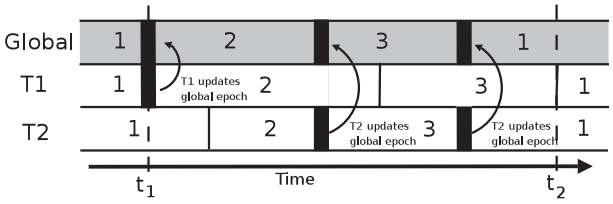

Figure 4. Illustration of EBR.

We address this deficit by comparing three recent reclamation schemes, showing the respective strengths and weaknesses of each. In Sections 2 and 3, we review these schemes and describe factors affecting their performance. Section 4 explains our experimental setup. Our analysis, in Section 5, reveals substantial performance differences between these schemes, the greatest source of which is per-operation atomic instructions. In Section 6, we discuss the relevance of our work to designers and implementers. We show that lockless algorithms and reclamation schemes are mostly independent, by combining a blocking reclamation scheme and a non-blocking algorithm, then comparing this combination to a fully non-blocking equivalent. We also present a new reclamation scheme that combines aspects of two other schemes to give good performance and ease of use. We close with a discussion of related work in Section 7 and summarize our conclusions in Section 8.

2 Memory Reclamation Schemes

This section briefly reviews the reclamation schemes we consider: quiescent-state-based reclamation (QSBR) [22, 2], epoch-based reclamation (EBR) [6], hazard-pointer-based reclamation (HPBR) [23, 24], and reference counting [29, 26]. We provide an overview of each scheme to help the reader understand our work; further details are available in the papers cited.

2.1 Blocking Schemes

We discuss two blocking schemes, QSBR and EBR, which use the concept of a *grace period*. A *grace period* is a time interval *[a,b]* such that, after time *b*, all nodes removed before time *a* may safely be reclaimed. These methods force threads to wait for other threads to complete their lockless operations in order for a grace period to occur. Failed or delayed threads can therefore prevent memory from being reclaimed. Eventually, memory allocation will fail, causing threads to block.

2.1.1 Quiescent-State-Based Reclamation (QSBR)

QSBR uses quiescent states to detect grace periods. A *quiescent state* for thread *T* is a state in which *T* holds no references to shared nodes; hence, a grace period for QSBR is any interval of time during which all threads pass through at least one quiescent state. The choice of quiescent states is application-dependent. Many operating system kernels contain natural quiescent states, such as voluntary context switch, and use QSBR to implement *read-copy update* (RCU) [22, 20, 7].

Figure 2 illustrates the relationship between quiescent states and grace periods in QSBR. Thread $T1$ goes through quiescent states at times t_1 and t_5, $T2$ at times t_2 and t_4, and $T3$ at time t_3. Hence, a grace period is any time interval containing either $[t_1, t_3]$ or $[t_3, t_5]$.

QSBR must detect grace periods so that removed nodes may be reclaimed; however, detecting minimal grace periods is unnecessary. In Figure 2, for example, *any* interval containing $[t_1, t_3]$ or $[t_3, t_5]$ is a quiescent state; implementations which check for grace periods only when threads enter quiescent states would detect $[t_1, t_5]$, since $T1$'s two quiescent states form the only pair from a single thread which enclose a grace period.

Figure 3 shows why QSBR is blocking. Here, $T2$ goes through no quiescent states (for example, due to blocking on I/O). Threads $T1$ and $T3$ are then prevented from reclaiming memory, since no grace periods exist. The ensuing memory exhaustion will eventually block all threads.

2.1.2 Epoch-Based Reclamation (EBR)

Fraser's EBR [6] follows QSBR in using grace periods, but uses *epochs* in place of QSBR's quiescent states. Each thread executes in one of three logical epochs, and may lag

```
 1 int search (struct list *l, long key)
 2 {
 3   node_t *cur;
 4   critical_enter();
 5   for (cur = l->list_head;
 6        cur != NULL; cur = cur->next) {
 7     if (cur->key >= key) {
 8       critical_exit();
 9       return (cur->key == key);
10     }
11   }
12   critical_exit();
13   return (0);
14 }
```

Figure 5. EBR concurrently-readable search.

at most one epoch behind the *global epoch*. Each thread atomically sets a per-thread flag upon entry into a *critical region*, indicating that the thread intends to access shared data without locks. Upon exit, the thread atomically clears its flag. No thread is allowed to access an EBR-protected object outside of a critical region.

Figure 4 shows how EBR tracks epochs, allowing safe memory reclamation. Upon entering a critical region, a thread updates its local epoch to match the global epoch. Hence, if the global epoch is *e*, threads in critical regions can be in either epoch *e* or *e-1*, but not *e+1* (all mod 3). Any node a thread removes during a given epoch may be safely reclaimed the next time the thread re-enters that epoch. Thus, the time period $[t_1, t_2]$ in the figure is a grace period. A thread will attempt to update the global epoch only if it has not changed for some pre-determined number of critical region entries.

As with QSBR, reclamation can be stalled by threads which fail in critical regions, but threads not in critical regions cannot stall EBR. EBR's bookkeeping is invisible to the applications programmer, making it easy for a programmer to use; however, Section 5 shows that this property imposes significant overhead on EBR.

Figure 5 shows an example of a search of a linked list which allows lockless reads but uses locking for updates. QSBR omits lines 4 and 12, which handle EBR's epoch bookkeeping, but is otherwise identical; QSBR's quiescent states are flagged explicitly at a higher level (see Figure 8).

2.2 Non-blocking schemes

This section presents the non-blocking reclamation schemes we evaluate: hazard-pointer-based reclamation (HPBR) and lock-free reference counting (LFRC).

2.2.1 Hazard-Pointer-Based Reclamation (HPBR)

Michael's HPBR [23] provides an existence locking mechanism for dynamically-allocated nodes. Each thread performing lockless operations has *K* hazard pointers which it uses to protect nodes from reclamation by other threads;

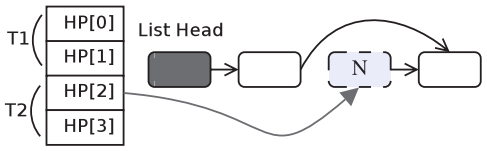

Figure 6. Illustration of HPBR.

```
 1 int search (struct list *l, long key)
 2 {
 3   node_t **prev, *cur, *next;
 4   /* Index of our first hazard pointer. */
 5   int base = getTID()*K;
 6   /* Offset into our hazard pointer segment. */
 7   int off = 0;
 8 try_again:
 9   prev = &l->list_head;
10   for (cur = *prev; cur != NULL; cur = next & ~1) {
11     /* Protect cur with a hazard pointer. */
12     HP[base+off] = cur;
13     memory_fence();
14     if (*prev != cur)
15       goto try_again;
16     next = cur->next;
17     if (cur->key >= key)
18       return (cur->key == key);
19     prev = &cur->next;
20     off = (off+1) % K;
21   }
22   return (0);
23 }
```

Figure 7. HPBR concurrently-readable search.

hence, we have *H=NK* hazard pointers in total. *K* is data-structure-dependent, and often small. Queues and linked lists need *K=2* hazard pointers, while stacks require only *K=1*; however, we know of no upper bound on *K* for general tree- or graph-traversal algorithms.

After removing a node, a thread places that node in a private list. When the list grows to a predefined size *R*, the thread reclaims each node lacking a corresponding hazard pointer. Increasing *R* amortizes reclamation overhead across more nodes, but increases memory usage.

An algorithm using HPBR must identify all *hazardous references* — references to shared nodes that may have been removed by other threads or that are vulnerable to the ABA problem [24]. Such references require hazard pointers. The algorithm sets a hazard pointer, then checks for node removal; if the node has not been removed, then it may safely be referenced. As long as the hazard pointer references the node, HPBR's reclamation routine refrains from reclaiming it. Figure 6 illustrates the use of HPBR. Node *N* has been removed from the linked list, but cannot be reclaimed because *T2*'s hazard pointer *HP[2]* references it.

Figure 7, showing code adapted from Michael [24], demonstrates HPBR with a search algorithm corresponding to Figure 5. At most two nodes must be protected: the current node and its predecessor (*K=2*). The code removing

nodes, which is not shown here, uses the low-order bit of the `next` pointer as a flag. This guarantees that the validation step on line 14 will fail and retry in case of concurrent removal. Full details are given by Michael [24].

Herlihy et al. [15] presented a very similar scheme called Pass the Buck (PTB). Since HPBR and PTB have similar per-operation costs, we believe that our HPBR results apply to PTB as well.

2.2.2 Lock-Free Reference Counting (LFRC)

Lock-free reference counting (LFRC) is a well-known non-blocking garbage-collection technique. Threads track the number of references to nodes, reclaiming any node whose count is zero. Valois's LFRC scheme [29] (corrected by Michael and Scott [26]) uses CAS and fetch-and-add (FAA), and requires nodes to retain their type after reclamation. Sundell's scheme [28], based on Valois's, is wait-free. The scheme of Detlefs et al. [5] allows nodes' types to change upon reclamation, but requires double compare-and-swap (DCAS), which no current CPU supports.

Michael [24] showed that LFRC introduces overhead which often makes lockless algorithms perform worse than lock-based versions. We include some experiments with Valois' scheme to reproduce Michael's findings.

3 Reclamation Performance Factors

We categorize factors which can affect reclamation scheme performance; we vary these factors in Section 5.

3.1 Memory Consistency

Current literature on lock-free algorithms generally assumes a sequentially-consistent [18] memory model, which prohibits instruction reordering and globally orders memory references. For performance reasons, however, modern CPUs provide weaker memory consistency models in which ordering can be enforced only when needed via special *fence* instructions. Although fences are often omitted from pseudocode, they are expensive on most modern CPUs and must be included in realistic performance analyses.

HPBR, EBR, and LFRC require per-operation fences. HPBR, as shown in Figure 7, requires a fence between hazard-pointer setting and validation, thus one fence per visited node. LFRC also requires per-node fences, in addition to atomic instructions needed to maintain reference counts. EBR requires two fences per operation: one when setting a flag when entering a critical region, and one when clearing it upon exit. Since QSBR has no per-operation fences, its per-operation overhead can be very low.

3.2 Data Structures and Workload

Data structures differ in both the operations they provide, and in their common workloads. Queues are write-only, but linked lists and hash tables are often read-mostly [19]. Blocking schemes may perform poorly with update-heavy structures, since the risk of memory exhaustion is higher.

Conversely, non-blocking schemes may perform poorly with operations such as list or tree traversal which visit many nodes, since they require per-node fences.

3.3 Threads and Scheduling

We expect contention due to concurrent threads to be a minor source of reclamation overhead; however, for the non-blocking schemes, it could be unbounded in degenerate cases: readers forced to repeatedly restart their traversals must *repeatedly* execute fence instructions for every node.

Thread preemption, especially when there are more threads than processors, can adversely affecting blocking schemes. Descheduled threads can delay reclamation, potentially exhausting memory, particularly in update-heavy workloads. Longer scheduling quanta may increase the risk of this exhaustion.

3.4 Memory Constraints

Although lock-free memory allocators exist [25], many allocators use locking. Blocking methods will see greater lock contention because they must access a lock-protected global pool more frequently. Furthermore, if a thread is preempted while holding such a lock, other threads will block on memory allocation. The size of the global pool is finite, and governs the likelihood of a blocking scheme exhausting all memory. Only HPBR [23] provides a provable bound on the amount of unreclaimed memory; it should thus be less sensitive to these constraints.

4 Experimental Setup

We evaluated the memory reclamation strategies with respect to the factors outlined in Section 3 using commodity SMP systems with IBM POWER CPUs. This section provides details on these aspects of our experiments.

4.1 Data Structures Used

We tested the reclamation schemes on linked lists and queues. We used Michael's ordered lock-free linked list [24], which forbids duplicate keys, and coded our concurrently-readable lists similarly. Because linked lists permit arbitrary lengths and read-to-write ratios, we used

```
 1 while (parent's timer has not expired) {
 2   for i from 1 to 100 do {
 3     key = random key;
 4     op = random operation;
 5     d = data structure;
 6     op(d, key);
 7   }
 8   if (using QSBR)
 9     quiescent_state();
10 }
```

Figure 8. Per-thread test pseudocode.

them heavily in our experiments. Our lock-free queue follows Michael and Scott's design [24]. Queues allow evaluating QSBR on a write-only data structure, which no prior studies have done.

4.2 Test Program

In our tests, a parent thread creates N child threads, starts a timer, and stops the threads upon timer expiry. Child threads count the number of operations they perform, and the parent then calculates the average *execution time* per operation by dividing the duration of the test by the total number of operations. The *CPU time* is the execution time divided by the minimum of the number of threads and the number of processors. CPU time compensates for increasing numbers of CPUs, allowing us to focus on synchronization overhead. Our tests report the average of five trials.

Each thread runs repeatedly through the test loop shown in Figure 8 until the timer expires. QSBR tests place a quiescent state at the end of the loop. The probabilities of inserting and removing nodes are equal, keeping data-structure size roughly constant throughout a given run.

We vary the number of threads and nodes. For linked lists, we also vary the ratio of reads to updates. As shown in Figure 8, each thread performs 100 operations per quiescent state; hence, grace-period-related overhead is amortized across 100 operations. For EBR, each *op* in Figure 8 is a critical region; a thread attempts to update the global epoch whenever it has entered a critical region 100 times since the last update, again amortizing grace-period-related overhead across 100 operations. For HPBR, we amortized reclamation overhead over $R = 2H + 100$ node removals. For consistency, QSBR and EBR both used the *fuzzy barrier* [11] algorithm from Fraser's EBR implementation [6].

The code for our experiments is available at `http://www.cs.toronto.edu/~tomhart/perflab/ipdps06.tgz`.

4.3 Operating Environment

We performed our experiments on the two machines shown in Table 1. The last line of this table gives the combined costs of locking and then unlocking a spinlock.

Table 1. Characteristics of Machines

	XServe	IBM POWER
CPUs	2x 2.0GHz PowerPC G5	8x 1.45 GHz POWER4+
Kernel	Linux 2.6.8-1.ydl.7g5-smp	Linux 2.6.13 (kernel.org)
Fence	78ns (156 cycles)	76ns (110 cycles)
CAS	52ns (104 cycles)	59ns (86 cycles)
Lock	231ns (462 cycles)	243ns (352 cycles)

Our experiment implements threads using processes. Our memory allocator is similar to that of Bonwick [4]. Each thread has two freelists of up to 100 elements each, and can acquire more memory from a global non-blocking stack of freelists. This non-blocking allocator allowed us to study reclamation performance without considering pathological locking conditions discussed in Section 3.

We implemented CAS using POWER's `LL/SC` instructions (`larx` and `stcx`), and fences using the `eieio` instruction. Our spinlocks were implemented using CAS and fences. Our algorithms used exponential backoff [1] upon encountering conflicts.

4.4 Limitations of Experiment

Microbenchmarks are never perfect [17], however, they allow us to study reclamation performance by varying each of the factors outlined in Section 3 independently. Our results show that these factors significantly affect reclamation performance. In macrobenchmark experiments, it is more difficult to gain insight into the causes of performance differences, and to test the schemes comprehensively.

Some applications may not have natural quiescent states; furthermore, detecting quiescent states in other applications may be more expensive than it is in our experiments. Our QSBR implementation, for example, is faster than that used in the Linux kernel, due to the latter's need to support dynamic insertion and removal of CPUs, interrupt handlers, and real-time workloads.

Our HPBR experiments statically allocate hazard pointers. Although this is sufficient for our experiments, some algorithms, to the best of our knowledge, require unbounded numbers of hazard pointers.

Despite these limitations, we believe that our experiments thoroughly evaluate these schemes, and show when each scheme is and is not efficient.

5 Performance Analysis

We first investigate the base costs for the reclamation schemes: single-threaded execution on small data structures. We then show how workload, list traversal length, number of threads, and preemption affect performance.

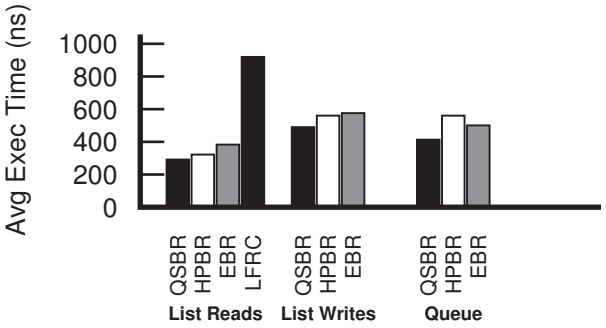

Figure 9. Base costs — single-threaded data from 8-CPU machine.

5.1 Base Costs

Figure 9 shows the single-threaded base costs of these schemes on non-blocking queues and single-element linked lists with no preemption or contention. We ran LFRC only on read-only workloads; these were sufficient for us to corroborate Michael's [24] result that LFRC performs poorly.

In these base cases, the dominant influence on performance is per-operation atomic instructions: compare-and-swap (CAS), fetch-and-add (FAA), and fences make LFRC much more expensive than the other schemes. Since EBR requires two fences per operation, and HPBR requires one for most operations considered here, EBR is usually the next most expensive. QSBR, needing no per-operation atomic instructions, is the cheapest scheme in the base case.

Workload affects the performance of these schemes. Under an update-intensive workload, a significant number of operations will involve removing nodes; for each attempt to reclaimed a removed node, HPBR must search the array of hazard pointers. This overhead can become significant for update-intensive workloads, as can be seen in Figure 9. We note that in our experiments, the performance of QSBR, EBR, and HPBR all increased linearly between read-only and update-only workloads.

5.2 Scalability with Traversal Length

Figure 10 shows the effect of list length on a single-threaded read-only workload. We observed similar results in write-only workloads. As expected, per-element fence instructions degrade HPBR's performance on long chains of elements; QSBR and EBR do much better.

Figure 11 shows the same scenario, but also includes LFRC. At best, LFRC takes more than twice as long as the next slowest scheme, and the performance gap rapidly increases with list length due to the multiple per-node atomic instructions. Because LFRC is always the worst scheme in terms of performance, we do not consider it further.

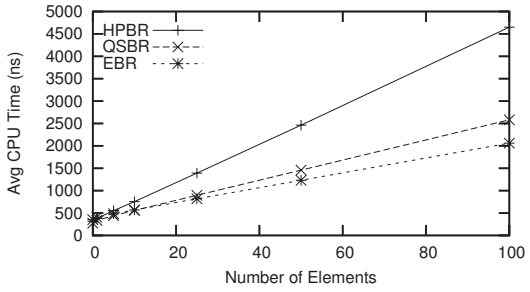

Figure 10. Effect of traversal length — read-only lock-free list, one thread, 8 CPUs.

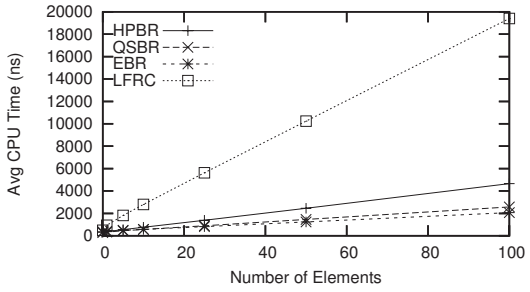

Figure 11. Effect of traversal length, including LFRC — read-only lock-free list, one thread, 8 CPUs.

5.3 Scalability with Threads

Concurrent performance is an obvious concern for memory reclamation schemes. We study the effect of threads sharing the data structure when there is no CPU contention, and when threads must also compete for the CPU.

5.3.1 No Preemption

To reduce the effects of CPU contention (thread preemption, migration, etc.), we use a maximum of seven threads, ensuring that one CPU is available for other processes, following Fraser [6].

Figures 12 and 13 show the performance of the reclamation schemes with a read-only workload on a linked list, and with a write-only workload on a queue. All three schemes scale almost linearly in the read-only case. In both cases, the schemes' relative performance seems to be unaffected by the number of threads.

5.3.2 With Preemption

To evaluate the performance of the reclamation schemes under preemption, we ran our tests on our 2-CPU machine, varying the number of threads from 1 to 32.

Figure 14 shows the performance of the schemes on a one-element lock-free linked list with a read-only workload.

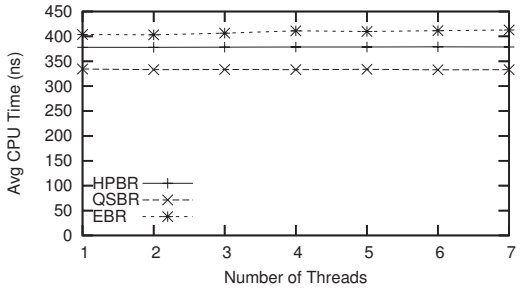

Figure 12. Effect of adding threads — read-only lock-free list, one element, 8 CPUs.

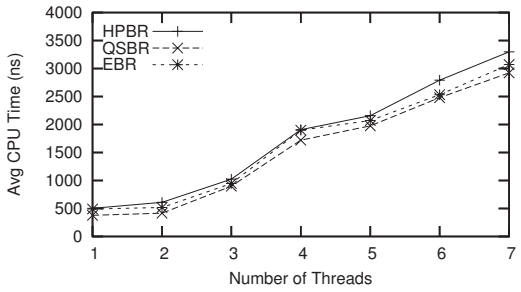

Figure 13. Effect of adding threads — lock-free queue, 8 CPUs.

This case eliminates reclamation overhead, focusing solely on read-side and fuzzy barrier overhead. In this case, the algorithms all scale well, with QSBR remaining cheapest. For the write-heavy workloads shown in Figure 15, HPBR performs best due to its non-blocking design.

The blocking schemes perform well on this write-heavy workload only because threads yield the processor upon allocation failure. Figure 16 shows the same test as Figure 15, but with busy-waiting upon allocation failure. Here, HPBR performs well, but EBR and QSBR quickly exhaust the pool of free memory. Each thread spins waiting for more memory to become free, thereby preventing grace periods from completing in a timely manner and hence delaying memory

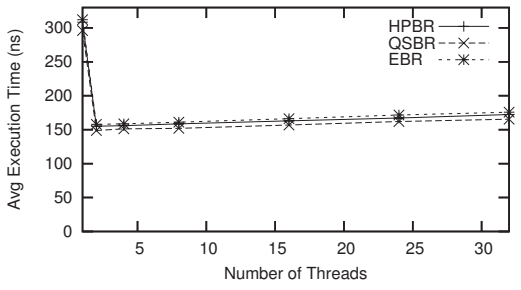

Figure 14. Effect of preemption — read-only lock-free list, 2 CPUs.

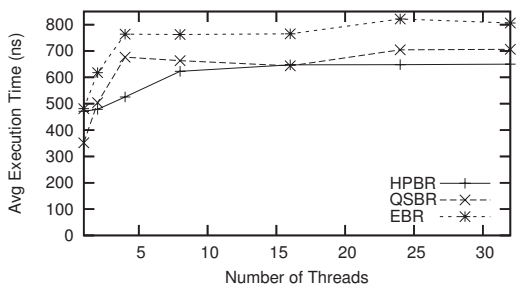

Figure 15. Effect of preemption — lock-free queue, 2 CPUs.

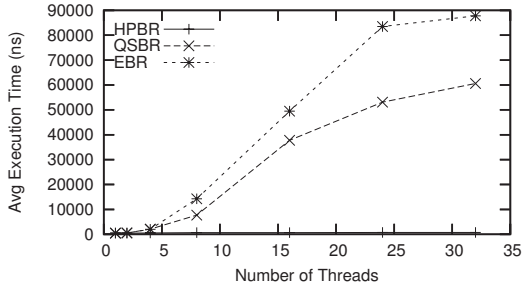

Figure 16. Effect of busy waiting when out of memory — lock-free queue, 2 CPUs.

reclamation.

Although this busy waiting would be a poor design choice in a real application, this test demonstrates that preemption and write-heavy workloads can cause QSBR and EBR to exhaust all memory. Similarly, Sarma and McKenney [27] have shown how QSBR-based components of the Linux kernel are vulnerable to denial-of-service attacks. Although this can be avoided with engineering effort – and has been, in Linux – it is in these situations that HPBR's fault-tolerance becomes valuable.

5.4 Summary

We note several trends of interest. First, in the base case, atomic instructions such as fences are the dominant cost. Second, when large numbers of elements must be traversed, HPBR and reference counting suffer from significant overhead due to extra atomic instructions. QSBR and EBR perform poorly when grace periods are stalled and the workload is update-intensive.

6 Consequences of Analysis

We describe the consequences of our analysis for comparing algorithms, designing new reclamation schemes, and choosing reclamation schemes for applications.

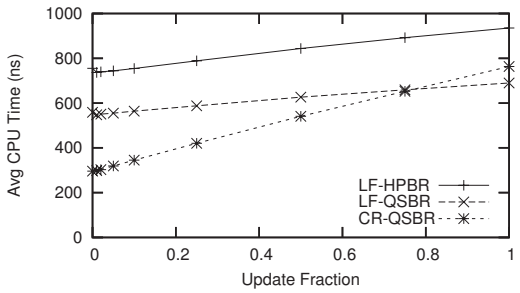

Figure 17. Lock-free versus concurrently-readable algorithms — ten-element lists, one thread, 8 CPUs.

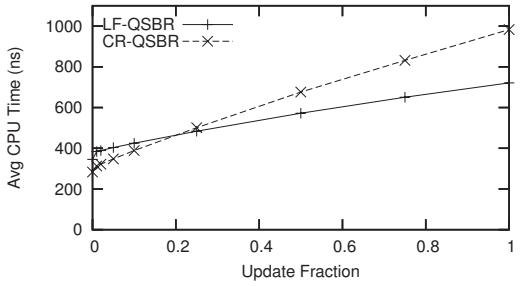

Figure 18. Lock-free versus concurrently-readable algorithms — hash tables with load factor 1, four threads, 8 CPUs.

6.1 Fair Evaluation of Algorithms

Reclamation schemes have profound performance effects that must be accounted for when experimentally evaluating new lockless algorithms.

For example, one of our early experiments compared a concurrently-readable linked list with QSBR, as is used in the Linux kernel, with a lock-free HPBR equivalent. Our intuition was that lock-free algorithms might pay a performance penalty for their fault-tolerance properties, as they do in read-mostly situations. The LF-HPBR and CR-QSBR traces in Figure 17 might lead to the erroneous conclusion that the concurrently-readable algorithm is always faster. A better analysis takes the LF-QSBR trace into account, noting that as the update fraction increases, lock-free performance improves, since its updates require fewer atomic instructions than does locking. This example shows that one can accurately compare two lockless algorithms only when each is using the same reclamation scheme.

LF-QSBR's higher per-node overhead makes it more attractive when there are fewer nodes. Figure 18 shows the performance of hash tables consisting of arrays of LF-QSBR or CR-QSBR single-element lists being concurrently accessed by four threads. For clarity, we omit HPBR from this graph — our intent is to compare the lock-free and

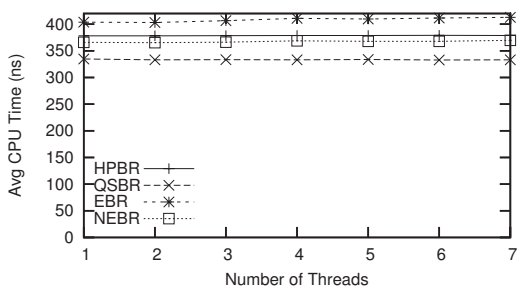

Figure 19. Performance of NEBR — lock-free list, 8 CPUs, read-only workload, variable number of threads.

concurrently-readable algorithms using a common reclamation scheme. Here, the lock-free algorithm out-performs locking for update fractions above about 15%. Lock-free lists and hash might therefore be practical for update-heavy situations in environments providing QSBR, such as OS kernels like Linux.

New reclamation schemes should also be evaluated by varying each of the factors that can affect their performance. For example, Gidenstam et al. [8] recently proposed a new non-blocking reclamation scheme that combines reference counting with HPBR, and can be proven to have several attractive properties. However, like HPBR and reference counting, it requires expensive per-node atomic operations. The evaluation of this scheme consisted only of experiments on double-ended queues, thus failing to evaluate scalability with data-structure size, an HPBR weakness. This failing shows the value of our analysis: it is necessary to vary the experimental parameters we considered to gain a full understanding of a given scheme's performance.

6.2 Improving Reclamation Performance

Improved reclamation schemes can be designed based on an understanding of the factors that affect performance. For example, we observe that a key difference between QSBR and EBR is the per-operation overhead of EBR's two fences. This observation allows us to make a modest improvement to EBR called *new epoch-based-reclamation* (NEBR).

NEBR requires compromising EBR's application-independence. Instead of setting and clearing the flag at the start and end of every lockless operation, we set it at the application level before entering any code that might contain NEBR critical regions. Since our flag is set and cleared at the application level, we can amortize the overhead of the corresponding fence instructions over a larger number of operations. We reran the experiment shown in Figure 12, but including NEBR, and found that NEBR scaled linearly and performed slightly better than did HPBR. Furthermore,

NEBR does not need the expensive per-node atomic operations that ruin HPBR's performance for long traversals.

NEBR is attractive because it is almost as fast as QSBR, but does not require quiescent states. Interestingly, recent realtime variants of the Linux-kernel RCU also dispense with quiescent states [21]. Ongoing work is expected to substantially reduce realtime RCU read-side overhead.

It is interesting to consider what sort of weakened non-blocking property, if any, could be defined such that one could create a corresponding reclamation scheme without requiring expensive per-node atomic operations.

6.3 Blocking Memory Reclamation for Non-Blocking Data Structures

We have shown that non-blocking data structures often perform better when using blocking reclamation schemes. One might question why one would want to use a non-blocking data structure in this case, since a halted would cause an infinite memory leak, thus destroying the non-blocking data structure's fault-tolerance guarantees.

However, non-blocking data structures are often used for reasons other than fault-tolerance; for example, Qprof [3] and Cache Kernel [10] both use such structures because they can be accessed from signal handlers without risk of self-deadlock. Blocking memory reclamation does not remove this benefit. In fact, Cache Kernel uses a blocking implementation of Type-Stable Memory to guard against read-reclamation races; its implementation [9] has similarities to QSBR. Non-blocking algorithms with blocking reclamation schemes similarly continue to benefit from resistance to preemption-induced convoying and priority inversion.

We view combining a non-blocking algorithm with a blocking reclamation scheme as part of a trend towards weakened non-blocking properties [16, 3], designed to preserve selected advantages of non-blocking synchronization while improving performance. In this case, threads have all the advantages of non-blocking synchronization, *unless* the system runs out of memory.

Conversely, one may also use non-blocking reclamation with blocking algorithms to reduce the amount of memory awaiting reclamation in face of preempted or failed threads.

7 Related Work

Relevant work on reclamation scheme design was discussed in Section 2. Previous work on the performance of these schemes, however, is limited.

Michael [24] criticized QSBR for its unbounded memory use, but did not compare the performance of QSBR to that of HPBR, or determine when this limitation affects a program.

Auslander implemented a lock-free hash table with QSBR in K42 [19]. No performance evaluation, either between different reclamation methods or between concurrently-readable and lock-free hash tables, was provided. We are unaware of any work combining QSBR with update-intensive non-blocking algorithms such as queues.

Fraser [6] noted, but did not thoroughly evaluate, HPBR's fence overhead, and used his EBR instead. Our work extends Fraser's, showing that EBR itself has high overhead, often exceeding that of HPBR.

8 Conclusions

We have performed the first fair comparison of blocking and non-blocking reclamation across a range of workloads, showing that reclamation has a huge effect on lockless algorithm performance. Choosing the right scheme for the environment in which a concurrent algorithm is expected to run is essential to having the algorithm perform well.

Our results show that QSBR is usually the best-performing reclamation scheme; however, the performance of both QSBR and EBR can suffer due to memory exhaustion in the face of thread preemption or failure. HPBR and EBR have higher base costs than QSBR due to their required fences. For EBR, the worst-case overhead of fences is constant, while for HPBR it is unbounded. HPBR and LFRC scale poorly when many nodes must be traversed.

Our analysis helped us to identify the main source of overhead in EBR and decrease it, resulting in NEBR. Furthermore, understanding the impact of reclamation schemes on algorithm performance enables fair comparison of different algorithms — in our case, lock-free and concurrently-readable lists and hash tables.

We reiterate that blocking reclamation can be useful with non-blocking algorithms: in the absence of thread failure, non-blocking algorithms still benefit from deadlock-freedom, signal handler safety, and avoidance of priority inversion. Nevertheless, the question of what sort of weakened non-blocking property could be satisfied by a reclamation scheme without the per-node overhead of current non-blocking reclamation scheme designs remains open.

9 Acknowledgments

We owe thanks to Maged Michael and Keir Fraser for helpful comments on their respective work, to Faith Fich and Cristiana Amza for much helpful advice, and to Dan Frye for his support of this effort. We are indebted to Martin Bligh, Andy Whitcroft, and the ABAT team for the use of the 8-CPU machine used in our experiments.

10 Legal Statement

References

[1] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Trans. Parallel and Distributed Syst.*, 1(1):6–16, 1990.

[2] A. Arcangeli, M. Cao, P. E. McKenney, and D. Sarma. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proc. USENIX Annual Technical Conf. (FREENIX Track)*, pages 297–310. USENIX Association, June 2003.

[3] H.-J. Boehm. An almost non-blocking stack. In *Proc. 23rd ACM Symp. on Principles of Distributed Computing*, pages 40–49, 2004.

[4] J. Bonwick and J. Adams. Magazines and Vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *USENIX Technical Conf., General Track*, pages 15–33, 2001.

[5] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[6] K. Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2004.

[7] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. 3rd Symp. on Operating Syst. Design and Impl.*, pages 87–100, 1999.

[8] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. In *Proc. 8th I-SPAN*, 2005.

[9] M. Greenwald. *Non-blocking Synchronization and System Design*. PhD thesis, Stanford University, 1999.

[10] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proc. 2nd USENIX Symp. on Operating Syst. Design and Impl.*, pages 123–136. ACM Press, 1996.

[11] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *Proc. 3rd Intl. Conf. on Arch. Support for Prog. Lang. and Operating Syst. (ASPLOS)*, pages 54–63, New York, NY, USA, 1989. ACM Press.

[12] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. 15th Intl. Conf. on Distributed Computing*, pages 300–314. Springer-Verlag, 2001.

[13] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):124–149, 1991.

[14] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Prog. Lang. Syst.*, 15(5):745–770, Nov. 1993.

[15] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proc. 16th Intl. Symp. on Distributed Computing*, Oct. 2002.

[16] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. 23rd Intl. Conf. on Distributed Computing Syst.*, pages 522–529. IEEE Computer Society, 2003.

[17] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh. Evaluating synchronization on shared address space multiprocessors: methodology and performance. In *Proc. SIGMETRICS 1999*, pages 23–34, New York, NY, USA, 1999. ACM Press.

[18] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[19] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[20] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *Ottawa Linux Symp.*, July 2001.

[21] P. E. McKenney and D. Sarma. Towards hard realtime response from the Linux kernel on SMP hardware. In *linux.conf.au*, Canberra, AU, April 2005.

[22] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Syst.*, pages 509–518, Las Vegas, NV, Oct. 1998.

[23] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proc. 21st ACM Symp. on Principles of Distributed Computing*, pages 21–30, July 2002.

[24] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. on Parallel and Distributed Syst.*, 15(6):491–504, June 2004.

[25] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 35–46, June 2004.

[26] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, Computer Science Department, University of Rochester, Dec. 1995.

[27] D. Sarma and P. E. McKenney. Issues with selected scalability features of the 2.6 kernel. In *Ottawa Linux Symp.*, July 2004.

[28] H. Sundell. Wait-free reference counting and memory management. In *Proc. 19th Intl. Parallel and Distributed Processing Symp.*, Apr. 2005.

[29] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 214–222, Aug. 1995.