To appear in OSDI 2006

Stasis: Flexible Transactional Storage

Russell Sears and Eric Brewer
University of California, Berkeley
{sears,brewer}@cs.berkeley.edu

An increasing range of applications requires robust support for atomic, durable and concurrent transactions. Databases provide the default solution, but force applications to interact via SQL and to forfeit control over data layout and access mechanisms. We argue there is a gap between DBMSs and file systems that limits designers of data-oriented applications.

Stasis is a storage framework that incorporates ideas from traditional write-ahead logging algorithms and file systems. It provides applications with flexible control over data structures, data layout, robustness, and performance. Stasis enables the development of unforeseen variants on transactional storage by generalizing write-ahead logging algorithms. Our partial implementation of these ideas already provides specialized (and cleaner) semantics to applications.

We evaluate the performance of a traditional transactional storage system based on Stasis, and show that it performs favorably relative to existing systems. We present examples that make use of custom access methods, modified buffer manager semantics, direct log file manipulation, and LSN-free pages. These examples facilitate sophisticated performance optimizations such as zero-copy I/O. These extensions are composable, easy to implement and significantly improve performance.

1 Introduction

As our reliance on computing infrastructure increases, a wider range of applications requires robust data management. Traditionally, data management has been the province of database management systems (DBMSs), which are well-suited to enterprise applications, but lead to poor support for systems such as web services, search engines, version systems, work-flow applications, bioinformatics, and scientific computing. These applications have complex transactional storage requirements, but do not fit well onto SQL or the monolithic approach of current databases. In fact, when performance matters these applications often avoid DBMSs and instead implement ad-hoc data management solutions [15, 17].

An example of this mismatch occurs with DBMS support for persistent objects. In a typical usage, an array of objects is made persistent by mapping each object to a row in a table (or sometimes multiple tables) [22] and then issuing queries to keep the objects and rows consistent. Also, for efficiency, most systems must buffer two copies of the application's working set in memory. This is an awkward and inefficient mechanism, and hence we claim that DBMSs do not support this task well.

Search engines and data warehouses in theory can use the relational model, but in practice need a very different implementation. Object-oriented, XML, and streaming databases all have distinct conceptual models and underlying implementations.

Scientific computing, bioinformatics and document management systems tend to preserve old versions and track provenance. Thus they each have a distinct conceptual model. Bioinformatics systems perform computations over large, semi-structured databases. Relational databases support none of these requirements well. Instead, office suites, ad-hoc text-based formats and Perl scripts are used for data management [48], with mixed success [57].

Our hypothesis is that 1) each of these areas has a distinct top-down conceptual model (which may not map well to the relational model); and 2) there exists a bottom-up layered framework that can better support all of these models and others.

To explore this hypothesis, we present Stasis, a library that provides transactional storage at a level of abstraction as close to the hardware as possible. It can support special-purpose transactional storage models in addition to ACID database-style interfaces to abstract data models. Stasis incorporates techniques from both databases (e.g. write-ahead logging) and operating systems (e.g. zero-copy techniques).

Our goal is to combine the flexibility and layering of

low-level abstractions typical for systems work with the complete semantics that exemplify the database field. By *flexible* we mean that Stasis can support a wide range of transactional data structures *efficiently*, and that it can support a variety of policies for locking, commit, clusters and buffer management. Also, it is extensible for new core operations and data structures. This flexibility allows it to support a wide range of systems and models.

By *complete* we mean full redo/undo logging that supports both *no force*, which provides durability with only log writes, and *steal*, which allows dirty pages to be written out prematurely to reduce memory pressure. By complete, we also mean support for media recovery, which is the ability to roll forward from an archived copy, and support for error-handling, clusters, and multithreading. These requirements are difficult to meet and form the *raison d'être* for Stasis: the framework delivers these properties as reusable building blocks for systems that implement complete transactions.

Through examples and their good performance, we show how Stasis efficiently supports a wide range of uses that fall in the gap between database and file system technologies, including persistent objects, graph- or XML-based applications, and recoverable virtual memory [42].

For example, on an object persistence workload, we provide up to a 4x speedup over an in-process MySQL implementation and a 3x speedup over Berkeley DB, while cutting memory usage in half (Section 5.3). We implemented this extension in 150 lines of C, including comments and boilerplate. We did not have this type of optimization in mind when we wrote Stasis, and in fact the idea came from a user unfamiliar with Stasis.

This paper begins by contrasting Stasis' approach with that of conventional database and transactional storage systems. It proceeds to discuss write-ahead logging, and describe ways in which Stasis can be customized to implement many existing (and some new) write-ahead logging variants. We present implementations of some of these variants and benchmark them against popular real-world systems. We conclude with a survey of related and future work.

An (early) open-source implementation of the ideas presented here is available (see Section 10).

2 Stasis is not a Database

Database research has a long history, including the development of many of the technologies we exploit. This section explains why databases are fundamentally inappropriate tools for system developers, and covers some of the previous responses of the systems community. These problems have been the focus of database and systems researchers for at least 25 years.

2.1 The Database View

The database community approaches the limited range of DBMSs by either creating new top-down models, such as object-oriented, XML or streaming databases [11, 28, 33], or by extending the relational model [14] along some axis, such as new data types [50]. We cover these attempts in more detail in Section 6.

An early survey of database implementations sought to enumerate the components used by database system implementors [4, 6]. This survey was performed due to difficulties in extending database systems into new application domains. It divided internal database routines into two broad modules: *conceptual mappings* and *physical database models*. It is the responsibility of a database implementor to choose a set of conceptual mappings that implement the desired higher-level abstraction (such as the relational model). The physical data model is chosen to support efficiently the set of mappings that are built on top of it.

A conceptual mapping based on the relational model might translate a relation into a set of keyed tuples. If the database were going to be used for short, write-intensive and high-concurrency transactions (e.g. banking), the physical model would probably translate sets of tuples into an on-disk B-tree. In contrast, if the database needed to support long-running, read-only aggregation queries over high-dimensional data (e.g. data warehousing), a physical model that stores the data in a sparse array format would be more appropriate [12, 58]. Although both kinds of databases are based upon the relational model they make use of different physical models in order to serve different classes of applications efficiently.

A basic claim of this paper is that no known physical data model can efficiently support the wide range of conceptual mappings that are in use today. In addition to sets, objects, and XML, such a model would need to cover search engines, version-control systems, workflow applications, and scientific computing, as examples. Similarly, a recent database paper argues that the "one size fits all" approach of DBMSs no longer works [51].

Instead of attempting to create such a unified model after decades of database research has failed to produce one, we opt to provide a bottom-up transactional toolbox that supports many models efficiently. This makes it easy for system designers to implement most data models that the underlying hardware can support, or to abandon the database approach entirely, and forgo a top-down model.

2.2 The Systems View

The systems community has also worked on this mismatch, which has led to many interesting projects. Examples include alternative durability models such as

QuickSilver [43], RVM [42], persistent objects [29], and persistent data structures [20, 32]. We expect that Stasis would simplify the implementation of most if not all of these systems. Section 6 covers these in more detail.

In some sense, our hypothesis is trivially true in that there exists a bottom-up framework called the "operating system" that can implement all of the models. A famous database paper argues that it does so poorly [49]. Our task is really to simplify the implementation of transactional systems through more powerful primitives that enable concurrent transactions with a variety of performance/robustness tradeoffs.

The closest system to ours in spirit is Berkeley DB, a highly successful lightweight alternative to conventional databases [45]. At its core, it provides the physical database model (relational storage system [3]) of a conventional database server. In particular, it provides transactional (ACID) operations on B-trees, hash tables, and other access methods. It provides flags that let its users tweak aspects of the performance of these primitives, and selectively disable the features it provides.

With the exception of the benchmark designed to compare the two systems, none of the Stasis applications presented in Section 5 are efficiently supported by Berkeley DB. This is a result of Berkeley DB's assumptions regarding workloads and low-level data representations. Thus, although Berkeley DB could be built on top of Stasis, Berkeley DB's data model and write-ahead logging system are too specialized to support Stasis.

3 Transactional Pages

This section describes how Stasis implements transactions that are similar to those provided by relational database systems, which are based on transactional pages. The algorithms described in this section are not novel, and are in fact based on ARIES [35]. However, they form the starting point for extensions and novel variants, which we cover in the next two sections.

As with other systems, Stasis' transactions have a multi-level structure. Multi-layered transactions were originally proposed as a concurrency control strategy for database servers that support high-level, application-specific extensions [55]. In Stasis, the lower level of an operation provides atomic updates to regions of the disk. These updates do not have to deal with concurrency, but must update the page file atomically, even if the system crashes.

Higher-level operations span multiple pages by atomically applying sets of operations to the page file, recording their actions in the log and coping with concurrency issues. The loose coupling of these layers lets Stasis' users compose and reuse existing operations.

3.1 Atomic Disk Operations

Transactional storage algorithms work by atomically updating portions of durable storage. These small atomic updates bootstrap transactions that are too large to be applied atomically. In particular, write-ahead logging (and therefore Stasis) relies on the ability to write entries to the log file atomically. Transaction systems that store sequence numbers on pages to track versions rely on atomic page writes in addition to atomic log writes.

In practice, a write to a disk page is not atomic (in modern drives). Two common failure modes exist. The first occurs when the disk writes a partial sector during a crash. In this case, the drive maintains an internal checksum, detects a mismatch, and reports it when the page is read. The second case occurs because pages span multiple sectors. Drives may reorder writes on sector boundaries, causing an arbitrary subset of a page's sectors to be updated during a crash. *Torn page detection* can be used to detect this phenomenon, typically by requiring a checksum for the whole page.

Torn and corrupted pages may be recovered by using *media recovery* to restore the page from backup. Media recovery works by reloading the page from an archive copy, and bringing it up to date by replaying the log.

For simplicity, this section ignores mechanisms that detect and restore torn pages, and assumes that page writes are atomic. We relax this restriction in Section 4.

3.2 Non-concurrent Transactions

This section provides the "Atomicity" and "Durability" properties for a single ACID transaction.[1] First we describe single-page transactions, then multi-page transactions. "Consistency" and "Isolation" are covered with concurrent transactions in the next section.

The insight behind transactional pages was that atomic page writes form a good foundation for full transactions. However, since page writes are no longer atomic, it might be better to think of these as transactional sectors.

The trivial way to achieve single-page transactions is to apply all of the updates to the page and then write it out on commit. The page must be pinned until commit to prevent write-back of uncommitted data, but no logging is required.

This approach performs poorly because we *force* the page to disk on commit, which leads to a large number of synchronous non-sequential writes. By writing redo information to the log before committing (write-ahead logging), we get *no-force* transactions and better performance, since the synchronous writes to the log are sequential. Later, the pages are written out asynchronously, often as part of a larger sequential write.

After a crash, we have to apply the redo entries to

those pages that were not updated on disk. To decide which updates to reapply, we use a per-page version number called the *log-sequence number* or *LSN*. Each update to a page increments the LSN, writes it on the page, and includes it in the log entry. On recovery, we load the page, use the LSN to figure out which updates are missing (those with higher LSNs), and reapply them.

Updates from aborted transactions should not be applied, so we also need to log commit records; a transaction commits when its commit record correctly reaches the disk. Recovery starts with an analysis phase that determines all of the outstanding transactions and their fate. The redo phase then applies the missing updates for committed transactions.

Pinning pages until commit also hurts performance, and could even affect correctness if a single transaction needs to update more pages than can fit in memory. A related problem is that with concurrency a single page may be pinned forever as long as it has at least one active transaction in progress all the time. Systems that support *steal* avoid these problems by allowing pages to be written back early. This implies we may need to undo updates on the page if the transaction aborts, and thus before we can write out the page we must write the undo information to the log.

On recovery, the redo phase applies all updates (even those from aborted transactions). Then, an undo phase corrects stolen pages for aborted transactions. Each operation that undo performs is recorded in the log, and the per-page LSN is updated accordingly. In order to ensure progress even with crashes during recovery, special log records mark which actions have been undone, so they may be skipped during recovery in the future. We also use these records, called *Compensation Log Records (CLRs)* to avoid undoing actions that we intend to keep even when transactions abort.

The primary difference between Stasis and ARIES for basic transactions is that Stasis allows user-defined operations, while ARIES defines a set of operations that support relational database systems. An *operation* consists of an undo and a redo function. Each time an operation is invoked, a corresponding log entry is generated. We describe operations in more detail in Section 3.4.

Given steal/no-force single-page transactions, it is relatively easy to build full transactions. To recover a multi-page transaction, we simply recover each of the pages individually. This works because steal/no-force completely decouples the pages: any page can be written back early (steal) or late (no-force).

3.3 Concurrent Transactions

Two factors make it more complicated to write operations that may be used in concurrent transactions. The first is familiar to anyone that has written multi-threaded code: Accesses to shared data structures must be protected by latches (mutexes). The second problem stems from the fact that abort cannot simply roll back physical updates. Fortunately, it is straightforward to reduce this second, transaction-specific problem to the familiar problem of writing multi-threaded software.

To understand the problems that arise with concurrent transactions, consider what would happen if one transaction, A, rearranges the layout of a data structure. Next, another transaction, B, modifies that structure and then A aborts. When A rolls back, its undo entries will undo the changes that it made to the data structure, without regard to B's modifications. This is likely to cause corruption.

Two common solutions to this problem are *total isolation* and *nested top actions*. Total isolation prevents any transaction from accessing a data structure that has been modified by another in-progress transaction. An application can achieve this using its own concurrency control mechanisms, or by holding a lock on each data structure until the end of the transaction (by performing *strict two-phase locking* on the entire data structure). Releasing the lock after the modification, but before the end of the transaction, increases concurrency. However, it means that follow-on transactions that use the data may need to abort if this transaction aborts (*cascading aborts*).

Nested top actions avoid this problem. The key idea is to distinguish between the logical operations of a data structure, such as adding an item to a set, and internal physical operations such as splitting tree nodes. The internal operations do not need to be undone if the containing transaction aborts; instead of removing the data item from the page, and merging any nodes that the insertion split, we simply remove the item from the set as application code would—we call the data structure's *remove* method. That way, we can undo the insertion even if the nodes that were split no longer exist, or if the data item has been relocated to a different page. This lets other transactions manipulate the data structure before the first transaction commits.

In Stasis, each nested top action performs a single logical operation by applying a number of physical operations to the page file. Physical redo and undo log entries are stored in the log so that recovery can repair any temporary inconsistency that the nested top action introduces. Once the nested top action has completed, a logical undo entry is recorded, and a CLR is used to tell recovery and abort to skip the physical undo entries.

This leads to a mechanical approach for creating reentrant, concurrent operations:

1. Wrap a mutex around each operation. With care, it is possible to use finer-grained latches in a Stasis operation [36], but it is rarely necessary.

2. Define a *logical* undo for each operation (rather than a set of page-level undos). For example, this is easy for a hash table: the undo for *insert* is *remove*. The logical undo function should arrange to acquire the mutex when invoked by abort or recovery.

3. Add a "begin nested top action" right after mutex acquisition, and an "end nested top action" right before mutex release. Stasis includes operations that provide nested top actions.

If the transaction that encloses a nested top action aborts, the logical undo will *compensate* for the effects of the operation, taking updates from concurrent transactions into account. Using this recipe, it is relatively easy to implement thread-safe concurrent transactions. Therefore, they are used throughout Stasis' default data structure implementations. This approach also works with the variable-sized atomic updates covered in Section 4.

3.4 User-Defined Operations

The first kind of extensibility enabled by Stasis is user-defined operations. Figure 1 shows how operations interact with Stasis. A number of default operations come with Stasis. These include operations that allocate and manipulate records, operations that implement hash tables, and a number of methods that add functionality to recovery. Many of the customizations described below are implemented using custom operations.

In this portion of the discussion, physical operations are limited to a single page, as they must be applied atomically. Section 4 removes this constraint.

Operations are invoked by registering a callback (the "operation implementation" in Figure 1) with Stasis at startup, and then calling `Tupdate()` to invoke the operation at runtime. Stasis ensures that operations follow the write-ahead logging rules required for steal/no-force transactions by controlling the timing and ordering of log and page writes.

The redo log entry consists of the LSN and an argument that will be passed to redo. The undo entry is analogous.[2] Each operation should be deterministic, provide an inverse, and acquire all of its arguments from the argument passed via `Tupdate()`, from the page it updates, or both. The callbacks used during forward operation are also used during recovery. Therefore operations provide a single redo function and a single undo function. There is no "do" function, which reduces the amount of recovery-specific code in the system.

The first step in implementing a new operation is to decide upon an external interface, which is typically cleaner than directly calling `Tupdate()` to invoke the operation(s). The externally visible interface is implemented by wrapper functions and read-only access methods. The wrapper function modifies the state of the page file by packaging the information that will be needed for redo/undo into a data format of its choosing. This data structure is passed into `Tupdate()`, which writes a log entry and invokes the redo function.

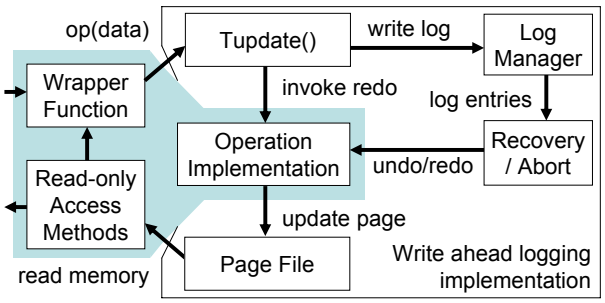

Figure 1: The portions of Stasis that directly interact with new operations. Arrows point in the direction of data flow.

The redo function modifies the page file directly (or takes some other action). It is essentially an interpreter for its log entries. Undo works analogously, but is invoked when an operation must be undone.

This pattern applies in many cases. In order to implement a "typical" operation, the operation's implementation must obey a few more invariants:

- Pages should only be updated inside physical redo and undo operation implementations.
- Logical operations may invoke other operations via `Tupdate()`. Recovery does not support logical redo, and physical operation implementations may not invoke `Tupdate()`.
- The page's LSN should be updated to reflect the changes (this is generally handled by passing the LSN to the page implementation).
- Nested top actions (and logical undo) or "big locks" (total isolation) should be used to manage concurrency (Section 3.3).

Although these restrictions are not trivial, they are not a problem in practice. Most read-modify-write actions can be implemented as user-defined operations, including common DBMS optimizations such as increment operations, and many optimizations based on ARIES [26, 36]. The power of Stasis is that by following these local restrictions, operations meet the global invariants required by correct, concurrent transactions.

Finally, for some applications, the overhead of logging information for redo or undo may outweigh their benefits. Operations that wish to avoid undo logging can call an API that pins the page until commit, and use an empty undo function. Similarly, forcing a page to be written out on commit avoids redo logging.

3.5 Application-specific Locking

The transactions described above provide the "Atomicity" and "Durability" properties of ACID. "Isolation" is typically provided by locking, which is a higher level but compatible layer. "Consistency" is less well defined but comes in part from low-level mutexes that avoid races, and in part from higher-level constructs such as unique key requirements. Stasis and most databases support this by distinguishing between *latches* and *locks*. Latches are provided using OS mutexes, and are held for short periods of time. Stasis' default data structures use latches in a way that does not deadlock. This allows higher-level code to treat Stasis as a conventional reentrant data structure library.

This section describes Stasis' latching protocols and describes two custom lock managers that Stasis' allocation routines use. Applications that want conventional transactional isolation (serializability) can make use of a lock manager or optimistic concurrency control [1, 27]. Alternatively, applications may follow the example of Stasis' default data structures, and implement deadlock prevention, or other custom lock management schemes.

Note that locking schemes may be layered as long as no legal sequence of calls to the lower level results in deadlock, or the higher level is prepared to handle deadlocks reported by the lower levels.

When Stasis allocates a record, it first calls a region allocator, which allocates contiguous sets of pages, and then it allocates a record on one of those pages. The record allocator and the region allocator each contain custom lock management. The lock management prevents one transaction from reusing storage freed by another, active transaction. If this storage were reused and then the transaction that freed it aborted, then the storage would be double-allocated.

The region allocator, which allocates large chunks infrequently, records the id of the transaction that created a region of freespace, and does not coalesce or reuse any storage associated with an active transaction. In contrast, the record allocator is called frequently and must enable locality. It associates a set of pages with each transaction, and keeps track of deallocation events, making sure that space on a page is never overbooked. Providing each transaction with a separate pool of freespace increases concurrency and locality. This is similar to Hoard [7] and McRT-malloc [23] (Section 6.4).

Note that both lock managers have implementations that are tied to the code they service, both implement deadlock avoidance, and both are transparent to higher layers. General-purpose database lock managers provide none of these features, supporting the idea that special-purpose lock managers are a useful abstraction. Locking schemes that interact well with object-oriented programming schemes [44] and exception handling [24] extend these ideas to larger systems.

Although custom locking is important for flexibility, it is largely orthogonal to the concepts described in this paper. We make no assumptions regarding lock managers being used by higher-level code in the remainder of this discussion.

4 LSN-free Pages

The recovery algorithm described above uses LSNs to determine the version number of each page during recovery. This is a common technique. As far as we know, it is used by all database systems that update data in place. Unfortunately, this makes it difficult to map large objects onto pages, as the LSNs break up the object. It is tempting to store the LSNs elsewhere, but then they would not be updated atomically, which defeats their purpose.

This section explains how we can avoid storing LSNs on pages in Stasis without giving up durable transactional updates. The techniques here are similar to those used by RVM [42], a system that supports transactional updates to virtual memory. However, Stasis generalizes the concept, allowing it to coexist with traditional pages and more easily support concurrent transactions.

In the process of removing LSNs from pages, we are able to relax the atomicity assumptions that we make regarding writes to disk. These relaxed assumptions allow recovery to repair torn pages without performing media recovery, and allow arbitrary ranges of the page file to be updated by a single physical operation.

Stasis' implementation does not currently support the recovery algorithm described in this section. However, Stasis avoids hard-coding most of the relevant subsystems. LSN-free pages are essentially an alternative protocol for atomically and durably applying updates to the page file. This will require the addition of a new page type that calls the logger to estimate LSNs; Stasis currently has three such types, and already supports the coexistence of multiple page types within the same page file or logical operation.

4.1 Blind Updates

Recall that LSNs were introduced to allow recovery to guarantee that each update is applied exactly once. This was necessary because some operations that manipulate pages are not idempotent, or simply make use of state stored in the page.

As described above, Stasis operations may make use of page contents to compute the updated value, and Stasis ensures that each operation is applied exactly once in the right order. The recovery scheme described in this section does not guarantee that operations will be applied

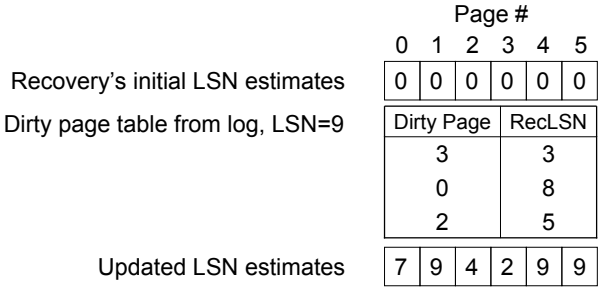

Figure 2: LSN estimation. If a page was not mentioned in the log, it must have been up-to-date on disk. RecLSN is the LSN of the entry that caused the page to become dirty. Subtracting one gives us a safe estimate of the page LSN.

exactly once, or even that they will be presented with a self-consistent version of a page during recovery.

Therefore, in this section we focus on operations that produce deterministic, idempotent redo entries that do not examine page state. We call such operations *blind updates*. For example, a blind update's operation could use log entries that contain a set of byte ranges with their new values. Note that we still allow code that invokes operations to examine the page file, just not during the redo phase of recovery.

Recovery works the same way as before, except that it now estimates page LSNs rather than reading them from pages. One safe estimate is the LSN of the most recent archive or log truncation point. Alternatively, Stasis could occasionally store its *dirty page table* to the log (Figure 2). The dirty page table lists all dirty pages and their *recovery LSNs*. It is used by ARIES to reduce the amount of work that must be performed during REDO.

The recovery LSN (RecLSN) is the LSN of the log entry that caused a clean (up-to-date on disk) page to become dirty. No log entries older than the RecLSN need to be applied to the page during redo. Therefore, redo can safely estimate the page LSN by choosing any number less than RecLSN. If a page is not in the table, redo can use the LSN of the log entry that contains the table, since the page must have been clean when the log entry was produced. Stasis writes the dirty page table to log whether or not LSN-free pages are in use, so we expect the runtime overhead to be negligible.

Although the mechanism used for recovery is similar, the invariants maintained during recovery have changed. With conventional transactions, if a page in the page file is internally consistent immediately after a crash, then the page will remain internally consistent throughout the recovery process. This is not the case with our LSN-free scheme. Internal page inconsistencies may be introduced because recovery has no way of knowing the exact version of a page. Therefore, it may overwrite new portions of a page with older data from the log. The page will then contain a mixture of new and old bytes, and any data structures stored on the page may be inconsistent. However, once the redo phase is complete, any old bytes will be overwritten by their most recent values, so the page will return to a self-consistent up-to-date state. (Section 4.4 explains this in more detail.)

Undo is unaffected except that any redo records it produces must be blind updates just like normal operation. We don't expect this to be a practical problem.

The rest of this section describes how concurrent, LSN-free pages allow standard file system and database optimizations to be easily combined, and shows that the removal of LSNs from pages simplifies recovery while increasing its flexibility.

4.2 Zero-copy I/O

We originally developed LSN-free pages as an efficient method for transactionally storing and updating multi-page objects, called *blobs*. If a large object is stored in pages that contain LSNs, then it is not contiguous on disk, and must be gathered together by using the CPU to do an expensive copy into a second buffer.

In contrast, modern file systems allow applications to perform a DMA copy of the data into memory, allowing the CPU to be used for more productive purposes. Furthermore, modern operating systems allow network services to use DMA and network-interface cards to read data from disk, and send it over the network without passing it through the CPU. Again, this frees the CPU, allowing it to perform other tasks.

We believe that LSN-free pages will allow reads to make use of such optimizations in a straightforward fashion. Zero-copy writes are more challenging, but the goal would be to use one sequential write to put the new version on disk and then update metadata accordingly. We need not put the blob in the log if we avoid update in place; most blob implementations already avoid update in place since the length may vary between writes. We suspect that contributions from log-based file systems [41] can address these issues. In particular, we imagine writing large blobs to a distinct log segment and just entering metadata in the primary log.

4.3 Concurrent RVM

LSN-free pages are similar to the recovery scheme used by recoverable virtual memory (RVM) and Camelot [16]. RVM used purely physical logging and LSN-free pages so that it could use `mmap` to map portions of the page file into application memory [42]. However, without support for logical log entries and nested top actions, it is difficult to implement a concurrent, durable data structure

using RVM or Camelot. (The description of Argus in Section 6.2.2 sketches one approach.)

In contrast, LSN-free pages allow logical undo and therefore nested top actions and concurrent transactions; a concurrent data structure need only provide Stasis with an appropriate inverse each time its logical state changes.

We plan to add RVM-style transactional memory to Stasis in a way that is compatible with fully concurrent in-memory data structures such as hash tables and trees, and with existing Stasis data structure implementations.

4.4 Unbounded Atomicity

Unlike transactions with per-page LSNs, transactions based on blind updates do not require atomic page writes and thus impose no meaningful boundaries on atomic updates. We still use pages to simplify integration into the rest of the system, but need not worry about torn pages. In fact, the redo phase of the LSN-free recovery algorithm effectively creates a torn page each time it applies an old log entry to a new page. However, it guarantees that all such torn pages will be repaired by the time redo completes. In the process, it also repairs any pages that were torn by a crash. This also implies that blind-update transactions work with storage technologies with different (and varying or unknown) units of atomicity.

Instead of relying upon atomic page updates, LSN-free recovery relies on a weaker property, which is that each bit in the page file must be either:

1. The version that was being overwritten at the crash.
2. The newest version of the bit written to storage.
3. Detectably corrupt (the storage hardware issues an error when the bit is read).

Modern drives provide these properties at a sector level: Each sector is updated atomically, or it fails a checksum when read, triggering an error. If a sector is found to be corrupt, then media recovery can be used to restore the sector from the most recent backup.

To ensure that we correctly update all of the old bits, we simply play the log forward from a point in time that is known to be older than the LSN of the page (which we must estimate). For bits that are overwritten, we end up with the correct version, since we apply the updates in order. For bits that are not overwritten, they must have been correct before and remain correct after recovery. Since all operations performed by redo are blind updates, they can be applied regardless of whether the initial page was the correct version or even logically consistent.

Figure 3 describes a page that is torn during crash, and the actions performed by redo that repair it. Assume that the initial version of the page, with LSN 0, is on disk, and the OS is in the process of writing out the version with

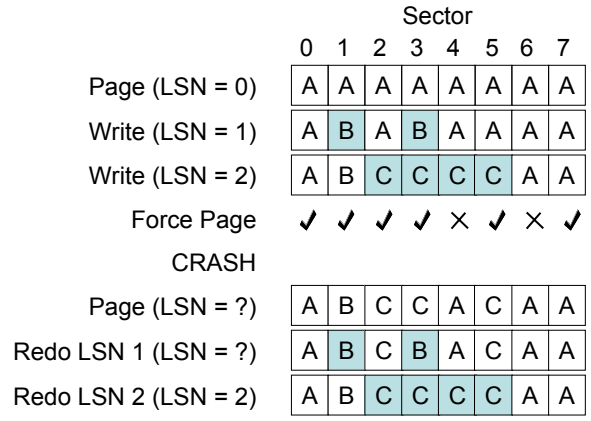

Figure 3: Torn pages and LSN-free recovery. The page is torn during the crash, but consistent once redo completes. Overwritten sectors are shaded.

LSN 2 when the system crashes. When recovery reads the page from disk, it may encounter any combination of sectors from these two versions.

Note that sectors zero, six and seven are not overwritten by any of the log entries that Redo will play back. Therefore, their values are unchanged in both versions of the page. In the example, zero and seven are overwritten during the crash, while six is left over from the old version of the page.

Redoing LSN 1 is unnecessary, since all of its sectors happened to make it to disk. However, recovery has no way of knowing this and applies the entry to the page, replacing sector three with an older version. When LSN 2 is applied, it brings this sector up to date, and also overwrites sector four, which did not make it to disk. At this point, the page is internally consistent.

Since LSN-free recovery only relies upon atomic updates at the bit level, it decouples page boundaries from atomicity and recovery. This allows operations to manipulate atomically (potentially non-contiguous) regions of arbitrary size by producing a single log entry. If this log entry includes a logical undo function (rather than a physical undo), then it can serve the purpose of a nested top action without incurring the extra log bandwidth of storing physical undo information. Such optimizations can be implemented using conventional transactions, but they appear to be easier to implement and reason about when applied to LSN-free pages.

4.5 Summary

In these last two sections, we explored some of the flexibility of Stasis. This includes user-defined operations, combinations of steal and force on a per-operation basis, flexible locking options, and a new class of transactions based on blind updates that enables better support for DMA, large objects, and multi-page operations. In

the next section, we show through experiments how this flexibility enables important optimizations and a wide-range of transactional systems.

5 Experiments

Stasis provides applications with the ability to customize storage routines and recovery semantics. In this section, we show that this flexibility does not come with a significant performance cost for general-purpose transactional primitives, and show how a number of special-purpose interfaces aid in the development of higher-level code while significantly improving application performance.

5.1 Experimental setup

We chose Berkeley DB in the following experiments because it provides transactional storage primitives similar to Stasis, is commercially maintained and is designed for high performance and high concurrency. For all tests, the two libraries provide the same transactional semantics unless explicitly noted.

All benchmarks were run on an Intel Xeon 2.8 GHz processor with 1GB of RAM and a 10K RPM SCSI drive using ReiserFS [40].[3] All results correspond to the mean of multiple runs with a 95% confidence interval with a half-width of 5%.

Our experiments use Berkeley DB 4.2.52 with the flags DB_TXN_SYNC (force log to disk on commit), and DB_THREAD (thread safety) enabled. We increased Berkeley DB's buffer cache and log buffer sizes to match Stasis' default sizes. If Berkeley DB implements a feature that Stasis is missing we enable it if it improves performance.

We disable Berkeley DB's lock manager for the benchmarks, though we use "Free Threaded" handles for all tests. This significantly increases performance by eliminating transaction deadlock, abort, and repetition. However, disabling the lock manager caused concurrent Berkeley DB benchmarks to become unstable, suggesting either a bug or misuse of the feature. With the lock manager enabled, Berkeley DB's performance in the multi-threaded benchmark (Section 5.2) strictly decreased with increased concurrency.

We expended a considerable effort tuning Berkeley DB and our efforts significantly improved Berkeley DB's performance on these tests. Although further tuning by Berkeley DB experts would probably improve Berkeley DB's numbers, we think our comparison shows that the systems' performance is comparable. As we add functionality, optimizations, and rewrite modules, Stasis' relative performance varies. We expect Stasis' extensions and custom recovery mechanisms to continue to perform similarly to comparable monolithic implementations.

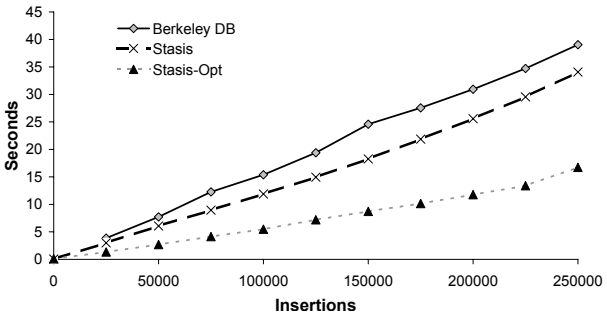

Figure 4: Performance of Stasis and Berkeley DB hash table implementations. The test is run as a single transaction, minimizing synchronous log writes.

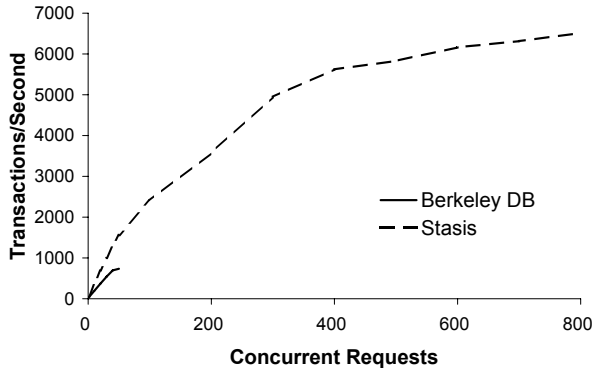

Figure 5: High-concurrency hash table performance. Our Berkeley DB test can only support 50 threads (see text).

5.2 Linear hash table

This section presents two hash table implementations built on top of Stasis, and compares them with the hash table provided by Berkeley DB. One of the Stasis implementations is simple and modular, while the other is monolithic and hand-tuned. Our experiments show that Stasis' performance is competitive, both with single-threaded and high-concurrency transactions.

The modular hash table uses nested top actions to update its internal structure atomically. It uses a *linear* hash function [30], allowing it to increase capacity incrementally. It is based on a number of modular subcomponents. Notably, the physical location of each bucket is stored in a growable array of fixed-length entries. This data structure is similar to Java's ArrayList. The bucket lists can be provided by either of Stasis' two linked list implementations. The first provides fixed-length entries, yielding a hash table with fixed-length keys and values. Our experiments use the second implementation, which provides variable-length entries (and therefore variable-length keys and values).

The hand-tuned hash table is also built on Stasis and also uses a linear hash function. However, it is monolithic and uses carefully ordered writes to reduce runtime

overheads such as log bandwidth. Berkeley DB's hash table is a popular, commonly deployed implementation, and serves as a baseline for our experiments.

Both of our hash tables outperform Berkeley DB on a workload that populates the tables by repeatedly inserting $(key, value)$ pairs (Figure 4). The performance of the modular hash table shows that data structure implementations composed from simpler structures can perform comparably to the implementations included in existing monolithic systems. The hand-tuned implementation shows that Stasis allows application developers to optimize important primitives.

Figure 5 describes the performance of the two systems under highly concurrent workloads using the ext3 file system.[4] For this test, we used the modular hash table, since we are interested in the performance of a simple, clean data structure implementation that a typical system implementor might produce, not the performance of our own highly tuned implementation.

Both Berkeley DB and Stasis can service concurrent calls to commit with a single synchronous I/O. Stasis scaled quite well, delivering over 6000 transactions per second, and provided roughly double Berkeley DB's throughput (up to 50 threads). Although not shown here, we found that the latencies of Berkeley DB and Stasis were similar.

5.3 Object persistence

Two different styles of object persistence have been implemented on top of Stasis. The first object persistence mechanism, pobj, provides transactional updates to objects in Titanium, a Java variant. It transparently loads and persists entire graphs of objects, but will not be discussed in further detail. The second variant was built on top of a C++ object persistence library, Oasys. Oasys uses plug-in storage modules that implement persistent storage, and includes plugins for Berkeley DB and MySQL. Like C++ objects, Oasys objects are explicitly freed. However, Stasis could also support concurrent and incremental atomic garbage collection [26].

This section describes how the Stasis plugin supports optimizations that reduce the amount of data written to log and halve the amount of RAM required. We present three variants of the Stasis plugin. The basic one treats Stasis like Berkeley DB. The "update/flush" variant customizes the behavior of the buffer manager. Finally, the "delta" variant uses update/flush, but only logs the differences between versions.

The update/flush variant allows the buffer manager's view of live application objects to become stale. This is safe since the system is always able to reconstruct the appropriate page entry from the live copy of the object. This reduces the number of times the plugin must update serialized objects in the buffer manager, and allows us to nearly eliminate the memory used by the buffer manager.

We implemented the Stasis buffer pool optimization by adding two new operations, update(), which updates the log when objects are modified, and flush(), which updates the page when an object is evicted from the application's cache.

The reason it would be difficult to do this with Berkeley DB is that we still need to generate log entries as the object is being updated. This would cause Berkeley DB to write data to pages, increasing the working set of the program and the amount of disk activity.

Furthermore, Stasis' copy of the objects is updated in the order objects are evicted from cache, not the update order. Therefore, the version of each object on a page cannot be determined from a single LSN.

We solve this problem by using blind updates to modify objects in place, but maintain a per-page LSN that is updated whenever an object is allocated or deallocated. At recovery, we apply allocations and deallocations based on the page LSN. To redo an update, we first decide whether the object that is being updated exists on the page. If so, we apply the blind update. If not, then the object must have been freed, so we do not apply the update. Because support for blind updates is only partially implemented, the experiments presented below mimic this behavior at runtime, but do not support recovery.

We also considered storing multiple LSNs per page and registering a callback with recovery to process the LSNs. However, in such a scheme, the object allocation routine would need to track objects that were deleted but still may be manipulated during redo. Otherwise, it could inadvertently overwrite per-object LSNs that would be needed during recovery. Alternatively, we could arrange for the object pool to update atomically the buffer manager's copy of all objects that share a given page.

The third plugin variant, "delta," incorporates the update/flush optimizations, but only writes changed portions of objects to the log. With Stasis' support for custom log formats, this optimization is straightforward.

Oasys does not provide a transactional interface. Instead, it is designed to be used in systems that stream objects over an unreliable network connection. The objects are independent of each other, so each update should be applied atomically. Therefore, there is never any reason to roll back an applied object update. Furthermore, Oasys provides a sync method, which guarantees the durability of updates after it returns. In order to match these semantics as closely as possible, Stasis' update/flush and delta optimizations do not write any undo information to the log. The Oasys sync method is implemented by committing the current Stasis transaction, and beginning a new one.

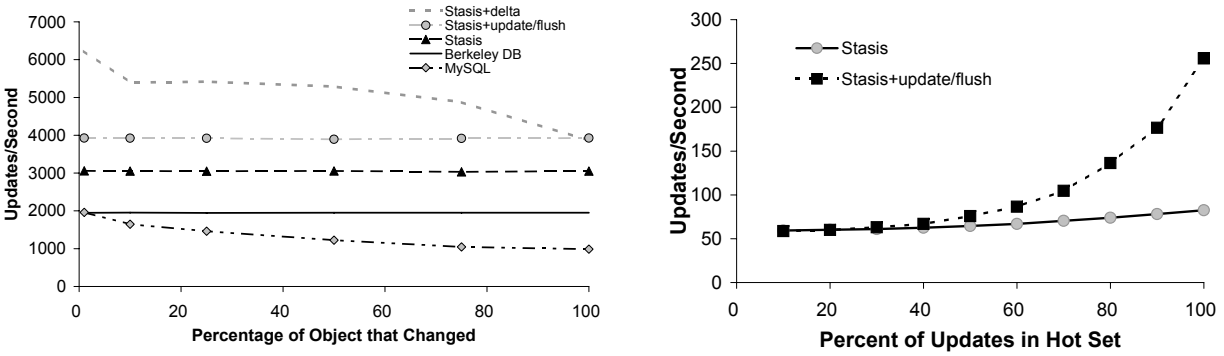

Figure 6: The effect of Stasis object-persistence optimizations under low and high memory pressure.

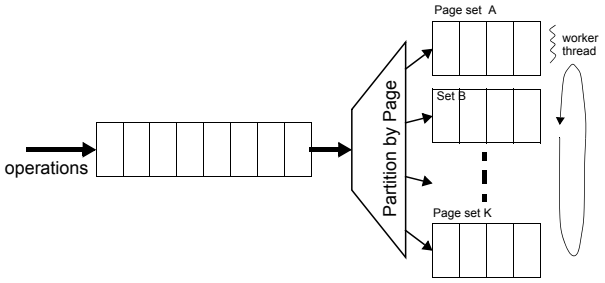

Figure 7: Locality-based request reordering. Requests are partitioned into queues. Queue are handled independently, improving locality and allowing requests to be merged.

As far as we can tell, MySQL and Berkeley DB do not support this optimization in a straightforward fashion. "Auto-commit" comes close, but does not quite provide the same durability semantics as Oasys' explicit syncs.

The operations required for the update/flush and delta optimizations required 150 lines of C code, including whitespace, comments and boilerplate function registrations.[5] Although the reasoning required to ensure the correctness of this optimization is complex, the simplicity of the implementation is encouraging.

In this experiment, Berkeley DB was configured as described above. We ran MySQL using InnoDB for the table engine. For this benchmark, it is the fastest engine that provides similar durability to Stasis. We linked the benchmark's executable to the `libmysqld` daemon library, bypassing the IPC layer. Experiments that used IPC were orders of magnitude slower.

Figure 6 presents the performance of the three Stasis variants, and the Oasys plugins implemented on top of other systems. In this test, none of the systems were memory bound. As we can see, Stasis performs better than the baseline systems, which is not surprising, since it exploits the weaker durability requirements.

In non-memory bound systems, the optimizations nearly double Stasis' performance by reducing the CPU overhead of marshalling and unmarshalling objects, and by reducing the size of log entries written to disk.

To determine the effect of the optimization in memory bound systems, we decreased Stasis' page cache size, and used O_DIRECT to bypass the operating system's disk cache. We partitioned the set of objects so that 10% fit in a *hot set*. Figure 6 also presents Stasis' performance as we varied the percentage of object updates that manipulate the hot set. In the memory bound test, we see that update/flush indeed improves memory utilization.

5.4 Request reordering

We are interested in enabling Stasis to manipulate sequences of application requests. By translating these requests into logical operations (such as those used for logical undo), we can manipulate and optimize such requests. Because logical operations generally correspond to application-level operations, application developers can easily determine whether logical operations may be reordered, transformed, or even dropped from the stream of requests that Stasis is processing. For example, requests that manipulate disjoint sets of data can be split across many nodes, providing load balancing. Requests that update the same piece of information can be merged into a single request; RVM's "log merging" implements this type of optimization [42]. Stream aggregation techniques and relational algebra operators could be used to transform data efficiently while it is laid out sequentially in non-transactional memory.

To experiment with the potential of such optimizations, we implemented a single-node request-reordering scheme that increases request locality during a graph traversal. The graph traversal produces a sequence of read requests that are partitioned according to their physical location in the page file. Partition sizes are chosen to fit inside the buffer pool. Each partition is processed until there are no more outstanding requests to read from it. The process iterates until the traversal is complete.

We ran two experiments. Both stored a graph of fixed-size objects in the growable array implementation that is

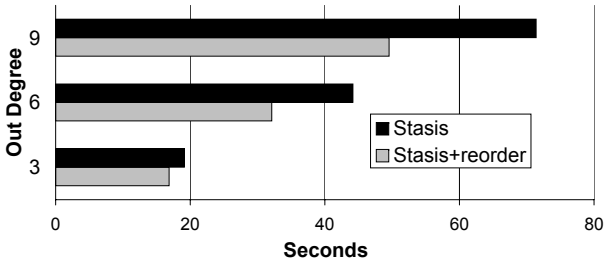

Figure 8: OO7 benchmark style graph traversal. The optimization performs well due to the presence of non-local nodes.

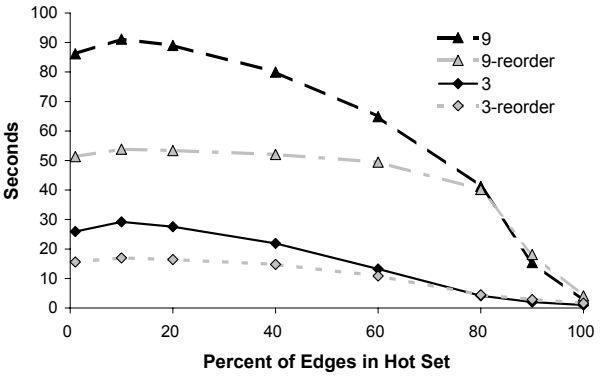

Figure 9: Hot-set based graph traversal for random graphs with out-degrees of 3 and 9. The multiplexer has low overhead, and improves performance when the graph has poor locality.

used as our linear hash table's bucket list. The first experiment (Figure 8) is loosely based on the OO7 database benchmark [9]. We hard-code the out-degree of each node and use a directed graph. Like OO7, we construct graphs by first connecting nodes together into a ring. We then randomly add edges until obtaining the desired out-degree. This structure ensures graph connectivity. Nodes are laid out in ring order on disk so at least one edge from each node is local.

The second experiment measures the effect of graph locality (Figure 9). Each node has a distinct hot set that includes the 10% of the nodes that are closest to it in ring order. The remaining nodes are in the cold set. We do not use ring edges for this test, so the graphs might not be connected. We use the same graphs for both systems.

When the graph has good locality, a normal depth-first search traversal and the prioritized traversal both perform well. As locality decreases, the partitioned traversal algorithm outperforms the naive traversal.

6 Related Work

6.1 Database Variations

This section discusses database systems with goals similar to ours. Although these projects were successful in many respects, each extends the range of a fixed abstract data model. In contrast, Stasis can support (in theory) any of these models and their extensions.

6.1.1 Extensible databases

Genesis is an early database toolkit that was explicitly structured in terms of the physical data models and conceptual mappings described above [5]. It allows database implementors to swap out implementations of the components defined by its framework. Like later systems (including Stasis), it supports custom operations.

Subsequent extensible database work builds upon these foundations. The Exodus [8] database toolkit is the successor to Genesis. It uses abstract data type definitions, access methods and cost models to generate query optimizers and execution engines automatically.

Object-oriented database systems [28] and relational databases with support for user-definable abstract data types (such as POSTGRES [52]) provide functionality similar to extensible database toolkits. In contrast to database toolkits, which leverage type information as the database server is compiled, object-oriented and object-relational databases allow types to be defined at runtime.

Both approaches extend a fixed high-level data model with new abstract data types. This is of limited use to applications that are not naturally structured in terms of queries over sets.

6.1.2 Modular databases

The database community is also aware of this gap. A recent survey [13] enumerates problems that plague users of state-of-the-art database systems. Essentially, it finds that modern databases are too complex to be implemented or understood as a monolithic entity. Instead, they have become unpredictable and unmanageable, preventing them from serving large-scale applications and small devices. Rather than concealing performance issues, SQL's declarative interface prevents developers from diagnosing and correcting underlying problems.

The study suggests that researchers and the industry adopt a highly modular "RISC" database architecture. This architecture would be similar to a database toolkit, but would standardize the interfaces of the toolkit's components. This would allow competition and specialization among module implementors, and distribute the effort required to build a full database [13].

Streaming applications face many of the problems that RISC databases could address. However, it is unclear whether a single interface or conceptual mapping would meet their needs. Based on experiences with their system, the authors of StreamBase argue that "one size fits all" database engines are no longer appropriate. Instead, they argue that the market will "fracture into a collection of independent...engines" [51]. This is in contrast to the RISC approach, which attempts to build a database in terms of interchangeable parts.

We agree with the motivations behind RISC databases and StreamBase, and believe they complement each other and Stasis well. However, our goal differs from these systems; we want to support applications that are a poor fit for database systems. As Stasis matures we hope that it will enable a wide range of transactional systems, including improved DBMSs.

6.2 Transactional Programming Models

Transactional programming environments provide semantic guarantees to the programs they support. To achieve this goal, they provide a single approach to concurrency and transactional storage. Therefore, they are complementary to our work; Stasis provides a substrate that makes it easier to implement such systems.

6.2.1 Nested Transactions

Nested transactions allow transactions to spawn subtransactions, forming a tree. *Linear* nesting restricts transactions to a single child. *Closed* nesting rolls children back when the parent aborts [37]. *Open* nesting allows children to commit even if the parent aborts.

Closed nesting uses database-style lock managers to allow concurrency within a transaction. It increases fault tolerance by isolating each child transaction from the others, and retrying failed transactions. (MapReduce is similar, but uses language constructs to statically enforce isolation [15].)

Open nesting provides concurrency between transactions. In some respect, nested top actions provide open, linear nesting, as the actions performed inside the nested top action are not rolled back when the parent aborts. (We believe that recent proposals to use open, linear nesting for software transactional memory will lead to a programming style similar to Stasis' [38].) However, logical undo gives the programmer the option to compensate for nested top actions. We expect that nested transactions could be implemented with Stasis.

6.2.2 Distributed Programming Models

Nested transactions simplify distributed systems; they isolate failures, manage concurrency, and provide durability. In fact, they were developed as part of Argus, a language for reliable distributed applications. An Argus program consists of guardians, which are essentially objects that encapsulate persistent and atomic data. Accesses to *atomic* data are serializable, while *persistent* data is atomic data that is stored on disk [29].

Originally, Argus only supported limited concurrency via total isolation, but was extended to support high concurrency data structures. Concurrent data structures are stored in non-atomic storage, but are augmented with information in atomic storage. This extra data tracks the status of each item stored in the structure. Conceptually, atomic storage used by a hash table would contain the values "Not present", "Committed" or "Aborted; Old Value = x" for each key in (or missing from) the hash. Before accessing the hash, the operation implementation would consult the appropriate piece of atomic data, and update the non-atomic data if necessary. Because the atomic data is protected by a lock manager, attempts to update the hash table are serializable. Therefore, clever use of atomic storage can be used to provide logical locking.

Efficiently tracking such state is not straightforward. For example, their hash table implementation uses a log structure to track the status of keys that have been touched by active transactions. Also, the hash table is responsible for setting policies regarding granularity and timing of disk writes [54]. Stasis operations avoid this complexity by providing logical undos, and by leaving lock management to higher-level code. This separates write-back and concurrency control policies from data structure implementations.

Camelot made a number of important contributions, both in system design, and in algorithms for distributed transactions [16]. It leaves locking to application level code, and updates data in place. (Argus uses shadow copies to provide atomic updates.) Camelot provides two logging modes: physical redo-only (no-steal, no-force) and physical undo/redo (steal, no-force). Because Camelot does not support logical undo, concurrent operations must be implemented similarly to those built with Argus. Camelot is similar to Stasis in that its low-level C interface is designed to enable multiple higher-level programming models, such as Avalon's C++ interface or an early version of RVM. However, like other distributed programming models, Camelot focuses on a particular class of distributed transactions. Therefore, it hard-codes assumptions regarding the structure of nested transactions, consensus algorithms, communication mechanisms, and so on.

More recent transactional programming schemes allow for multiple transaction implementations to cooperate as part of the same distributed transaction. For example, X/Open DTP provides a standard networking proto-

col that allows multiple transactional systems to be controlled by a single transaction manager [53]. Enterprise Java Beans is a standard for developing transactional middleware on top of heterogeneous storage. Its transactions may not be nested. This simplifies its semantics, and leads to many, short transactions, improving concurrency. However, flat transactions are somewhat rigid, and lead to situations where committed transactions have to be manually rolled back by other transactions [47]. The Open Multithreaded Transactions model is based on nested transactions, incorporates exception handling, and allows parents to execute concurrently with their children [24].

QuickSilver is a distributed transactional operating system. It provides a transactional IPC mechanism, and allows varying degrees of isolation, both to support legacy code, and to provide an appropriate environment for custom transactional software [21]. By providing an environment that allows multiple, independently written, transactional systems to interoperate, QuickSilver would complement Stasis nicely.

The QuickSilver project showed that transactions can meet the demands of most applications, provided that long-running transactions do not exhaust system resources, and that flexible concurrency control policies are available. Nested transactions are particularly useful when a series of program invocations form a larger logical unit [43].

Clouds is an object-oriented, distributed transactional operating system. It uses shared abstract types [44] and per-object atomicity specifications to provide concurrency control among the objects in the system [2]. These formalisms could be used during the design of high-concurrency Stasis operations.

6.3 Data Structure Frameworks

As mentioned in Sections 2.2 and 5, Berkeley DB is a system quite similar to Stasis, and gives application programmers raw access to transactional data structures such as a single-node B-Tree and hash table [45].

Cluster hash tables provide a scalable, replicated hash table implementation by partitioning the table's buckets across multiple systems [20]. Boxwood treats each system in a cluster of machines as a "chunk store," and builds a transactional, fault tolerant B-Tree on top of the chunks that these machines export [32].

Stasis is complementary to Boxwood and cluster hash tables; those systems intelligently compose a set of systems for scalability and fault tolerance. In contrast, Stasis makes it easy to push intelligence into the individual nodes, allowing them to provide primitives that are appropriate for the higher-level service.

6.4 Data layout policies

Data layout policies make decisions based upon assumptions about the application. Ideally, Stasis would allow application-specific layout policies to be used interchangeably, This section describes strategies for data layout that we believe Stasis could eventually support.

Some large object storage systems allow arbitrary insertion and deletion of bytes [10] within the object, while typical file systems provide append-only allocation [34]. Record-oriented allocation, such as in VMS Record Management Services [39] and GFS [18], breaks files into addressable units. Write-optimized file systems lay files out in the order they were written rather than in logically sequential order [41].

Schemes to improve locality among small objects exist as well. Relational databases allow users to specify the order in which tuples will be laid out, and often leave portions of pages unallocated to reduce fragmentation as new records are allocated.

Memory allocation routines such as Hoard [7] and McRT-malloc [23] address this problem by grouping allocated data by thread or transaction, respectively. This increases locality, and reduces contention created by unrelated objects stored in the same location. Stasis' current record allocator is based on these ideas (Section 3.5).

Allocation of records that must fit within pages and be persisted to disk raises concerns regarding locality and page layouts. Depending on the application, data may be arranged based upon hints [46], pointer values and write order [31], data type [25], or access patterns [56].

We are interested in allowing applications to store records in the transaction log. Assuming log fragmentation is kept to a minimum, this is particularly attractive on a single disk system. We plan to use ideas from LFS [41] and POSTGRES [52] to implement this.

7 Future Work

Complexity problems may begin to arise as we attempt to implement more extensions to Stasis. However, Stasis' implementation is still fairly simple:

- The core of Stasis is roughly 3000 lines of C code, and implements the buffer manager, IO, recovery, and other systems.
- Custom operations account for another 3000 lines.
- Page layouts and logging implementations account for 1600 lines.

The complexity of the core of Stasis is our primary concern, as it contains the hard-coded policies and assumptions. Over time, it has shrunk as functionality has

moved into extensions. We expect this trend to continue as development progresses.

A resource manager is a common pattern in system software design, and manages dependencies and ordering constraints among sets of components. Over time, we hope to shrink Stasis' core to the point where it is simply a resource manager that coordinates interchangeable implementations of the other components.

8 Conclusion

We presented Stasis, a transactional storage library that addresses the needs of system developers. Stasis provides more opportunities for specialization than existing systems. The effort required to extend Stasis to support a new type of system is reasonable, especially when compared to current practices, such as working around limitations of existing systems, breaking guarantees regarding data integrity, or reimplementing the entire storage infrastructure from scratch.

We demonstrated that Stasis provides fully concurrent, high-performance transactions, and explored how it can support a number of systems that currently make use of suboptimal or ad-hoc storage approaches. Finally, we described how Stasis can be extended in the future to support a larger range of systems.

9 Acknowledgements

Thanks to shepherd Bill Weihl for helping us present these ideas well, or at least better. The idea behind the Oasys buffer manager optimization is from Mike Demmer; he and Bowei Du implemented Oasys. Gilad Arnold and Amir Kamil implemented pobj. Jim Blomo, Jason Bayer, and Jimmy Kittiyachavalit worked on an early version of Stasis.

Thanks to C. Mohan for pointing out that per-object LSNs may be inadvertently overwritten during recovery. Jim Gray suggested we use a resource manager to track dependencies within Stasis and provided feedback on the LSN-free recovery algorithms. Joe Hellerstein and Mike Franklin provided us with invaluable feedback.

Portions of this work were performed at Intel Research Berkeley.

10 Availability

Additional information, and Stasis' source code is available at:

`http://www.cs.berkeley.edu/~sears/stasis/`

References

[1] AGRAWAL, R., CAREY, M. J., AND LIVNY, M. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems* (1987).

[2] ALLCHIN, J. E., AND MCKENDRY, M. S. Synchronization and recovery of actions. In *PODC* (1983), pp. 31–44.

[3] ASTRAHAN, M. ET AL. System R: Relational approach to database management. *ACM Transactions on Database Systems 1*, 2 (1976), 97–137.

[4] BATORY, D. S. Conceptual-to-internal mappings in commercial database systems. In *PODS* (1984), pp. 70–78.

[5] BATORY, D. S., BARNETT, J. R., GARZA, J. F., SMITH, K. P., TSUKUDA, K., TWICHELL, B. C., AND WISE, T. E. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering 14*, 11 (November 1988), 1711–1729.

[6] BATORY, D. S., AND GOTLIEB, C. C. A unifying model of physical databases. *ACM Transactions on Database Systems 7*, 4 (1982), 509–539.

[7] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. *ACM SIGPLAN Notices 35*, 11 (2000), 117–128.

[8] CAREY, M. J., DEWITT, D. J., FRANK, D., GRAEFE, G., MURALIKRISHNA, M., RICHARDSON, J., AND SHEKITA, E. J. The architecture of the EXODUS extensible DBMS. In *OODS* (1986), pp. 52–65.

[9] CAREY, M. J., DEWITT, D. J., AND NAUGHTON, J. F. The OO7 benchmark. In *SIGMOD* (1993), pp. 12–21.

[10] CAREY, M. J., DEWITT, D. J., RICHARDSON, J. E., AND SHEKITA, E. J. Object and file management in the EXODUS extensible database system. In *VLDB* (1986), pp. 91–100.

[11] CHANDRASEKARAN, S., AND FRANKLIN, M. Streaming queries over streaming data. In *VLDB* (2002).

[12] CHAUDHURI, S., AND DAYAL, U. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record 26*, 1 (1997), 65–74.

[13] CHAUDHURI, S., AND WEIKUM, G. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *VLDB* (2000).

[14] CODD, E. F. A relational model of data for large shared data banks. *Communications of the ACM 13*, 6 (June 1970), 377–387.

[15] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004).

[16] EPPINGER, J. L., MUMMERT, L. B., AND SPECTOR, A. Z., Eds. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

[17] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. Cluster-based scalable network services. In *SOSP* (1997), pp. 78–91.

[18] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The Google file system. In *SOSP* (2003), pp. 29–43.

[19] GRAY, J., AND REUTERS, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[20] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. E. Scalable, distributed data structures for Internet service construction. In *OSDI* (2000), pp. 319–332.

[21] HASKIN, R. L., MALACHI, Y., SAWDON, W., AND CHAN, G. Recovery management in QuickSilver. *ACM Transactions on Computer Systems 6*, 1 (1988), 82–108.

[22] Hibernate. http://www.hibernate.org/.

[23] HUDSON, R. L., SAHA, B., ADL-TABATABAI, A.-R., AND HERTZBERG, B. McRT-Malloc: A scalable transactional memory allocator. In *ISMM* (2006), pp. 74–83.

[24] KIENZLE, J., STROHMEIER, A., AND ROMANOVSKY, A. B. Open multithreaded transactions: Keeping threads and exceptions under control. In *WORDS* (2001), pp. 197–205.

[25] KIM, W., GARZA, J. F., BALLOU, N., AND WOELK, D. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering* (1990).

[26] KOLODNER, E. K., AND WEIHL, W. E. Atomic incremental garbage collection and recovery for a large stable heap. In *SIGMOD* (1993), pp. 177–186.

[27] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6*, 2 (June 1981), 213–226.

[28] LAMB, C., LANDIS, G., ORENSTEIN, J. A., AND WEINREB, D. The ObjectStore database system. *Communications of the ACM 34*, 10 (1991), 50–63.

[29] LISKOV, B. Distributed programming in Argus. *Communications of the ACM 31*, 3 (March 1988), 300–312.

[30] LITWIN, W. Linear hashing: A new tool for file and table addressing. In *VLDB* (1980), pp. 224–232.

[31] LOHMAN, G., LINDSAY, B., PIRAHESH, H., AND SCHIEFER, K. B. Extensions to Starburst: Objects, types, functions, and rules. *Communications of the ACM 34*, 10 (October 1991), 95–109.

[32] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI* (2004), pp. 105–120.

[33] MCHUGH, J., ABITEBOUL, S., GOLDMAN, R., QUASS, D., AND WIDOM, J. Lore: A database management system for semistructured data. *ACM SIGMOD Record 26*, 3 (September 1997), 54–66.

[34] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems* (1984).

[35] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. M. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems 17*, 1 (1992), 94–162.

[36] MOHAN, C., AND LEVINE, F. *ARIES/IM: An efficient and high concurrency index management method using write-ahead logging*. ACM Press, 1992.

[37] MOSS, J. E. B. *Nested transactions: An approach to reliable distributed computing*. MIT, 1985.

[38] MOSS, J. E. B. Open nested transactions: Semantics and support. In *WMPI* (2006).

[39] *OpenVMS Record Management Services Reference Manual*, June 2002.

[40] REISER, H. T. ReiserFS. http://www.namesys.com.

[41] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In *SOSP* (1992).

[42] SATYANARAYANAN, M., MASHBURN, H. H., KUMAR, P., STEERE, D. C., AND KISTLER, J. J. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems 12*, 1 (February 1994), 33–57.

[43] SCHMUCK, F. B., AND WYLLIE, J. C. Experience with transactions in QuickSilver. In *SOSP* (1991), pp. 239–253.

[44] SCHWARZ, P. M., AND SPECTOR, A. Z. Synchronizing shared abstract types. *ACM Transactions on Computer Systems 2*, 3 (April 1984), 223–250.

[45] SELTZER, M., AND OLSEN, M. LIBTP: Portable, modular transactions for UNIX. In *Usenix* (January 1992).

[46] SHEKITA, E. J., AND ZWILLING, M. J. Cricket: A mapped, persistent object store. In *POS* (1990), pp. 89–102.

[47] SILAGHI, R., AND STROHMEIER, A. Critical evaluation of the EJB transaction model. In *FIDJI* (2002), pp. 15–28.

[48] STEIN, L. How Perl saved the Human Genome Project. *Dr Dobb's Journal* (July 2001).

[49] STONEBRAKER, M. Operating system support for database management. *Communications of the ACM 24*, 7 (July 1981), 412–418.

[50] STONEBRAKER, M. Inclusion of new types in relational data base systems. In *ICDE* (1986), pp. 262–269.

[51] STONEBRAKER, M., AND ÇETINTEMEL, U. "One size fits all": An idea whose time has come and gone. In *ICDE* (2005), pp. 2–11.

[52] STONEBRAKER, M., AND KEMNITZ, G. The POSTGRES next-generation database management system. *Communications of the ACM 34*, 10 (October 1991), 79–92.

[53] THE OPEN GROUP. *Distributed Transaction Processing: Reference Model*, 1996.

[54] WEIHL, W., AND LISKOV, B. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems 7*, 2 (April 1985), 244–269.

[55] WEIKUM, G., AND SCHEK, H.-J. Architectural issues of transaction management in multi-layered systems. In *VLDB* (1984), pp. 454–465.

[56] YONG, V.-F., NAUGHTON, J. F., AND YU, J.-B. Storage reclamation and reorganization in client-server persistent object stores. In *ICDE* (1994), pp. 120–131.

[57] ZEEBERG, B. R., RISS, J., KANE, D. W., BUSSEY, K. J., UCHIO, E., LINEHANN, W. M., BARRETT, J. C., AND WEINSTEIN, J. N. Mistaken identifiers: Gene name errors can be introduced inadvertently when using Excel in bioinformatics. *BMC Bioinformatics* (2004).

[58] ZHAO, Y., DESHPANDE, P. M., AND NAUGHTON, J. F. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD* (1997), pp. 159–170.

Notes

[1] The "A" in ACID really means "atomic persistence of data," rather than "atomic in-memory updates," as the term is normally used in systems work; the latter is covered by "C" and "I" [19].

[2] For efficiency, undo and redo operations are packed into a single log entry. Both must take the same parameters.

[3] We found that the relative performance of Berkeley DB and Stasis under single-threaded testing is sensitive to file system choice, and we plan to investigate the reasons why the performance of Stasis under ext3 is degraded. However, the results relating to the Stasis optimizations are consistent across file system types.

[4] Multi-threaded benchmarks were performed using an ext3 file system. Concurrency caused both Berkeley DB and Stasis to behave unpredictably under ReiserFS. Stasis' multi-threaded throughput was significantly better than Berkeley DB's with both file systems.

[5] These figures do not include the simple LSN-free object logic required for recovery, as Stasis does not yet support LSN-free operations.