Stasys: System for adaptable, transactional storage

Russell Sears
UC Berkeley

Eric Brewer
UC Berkeley

An increasing range of applications require 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.

Stasys is a storage framework that incorporates ideas from traditional write-ahead-logging storage algorithms and file systems. It provides applications with flexible control over data structures, data layout, performance and robustness properties. Stasys 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 Stasys, 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 require 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, grid computing and scientific computing. These applications have complex transactional storage requirements but do not fit well onto SQL or the monolithic approach of current databases.

Simply providing access to a database system's internal storage module is an improvement. However, many of these applications require special transactional properties that general purpose transactional storage systems do not provide. In fact, DBMSs are often not used for these systems, which instead implement custom, ad-hoc data management tools on top of file systems.

A typical example of this mismatch is in the 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) [14] and then issuing queries to keep the objects and rows consistent. An update must confirm it has the current version, modify the object, write out a serialized version using the SQL update command and commit. Also, for efficiency, most systems must buffer two copies of the application's working set in memory. This is an awkward and slow mechanism.

Bioinformatics systems perform complex scientific computations over large, semi-structured databases with rapidly evolving schemas. Versioning and lineage tracking are also key concerns. Relational databases support none of these requirements well. Instead, office suites, ad-hoc text-based formats and Perl scripts are used for data management [25] (with mixed success [27]).

This paper presents Stasys, a library that provides transactional storage at a level of abstraction as close to the hardware as possible. The library can support special purpose, transactional storage interfaces in addition to ACID database-style interfaces to abstract data models. Stasys incorporates techniques from databases (e.g. write-ahead-logging) and 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 Stasys can implement a wide range of transactional data structures, that it can support a variety of policies for locking, commit, clusters and

buffer management. Also, it is extensible for new core operations and new data structures. It is this flexibility that allows the support of a wide range of systems.

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 Stasys: the framework delivers these properties as reusable building blocks for systems that implement complete transactions.

Through examples and their good performance, we show how Stasys supports a wide range of uses that fall in the gap between database and filesystem technologies, including persistent objects, graph or XML based applications, and recoverable virtual memory [23].

For example, on an object serialization 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 4.4).

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 Stasys. In fact, the idea came from a potential user that is not familiar with Stasys.

This paper begins by contrasting Stasys' approach with that of conventional database and transactional storage systems. It proceeds to discuss write-ahead-logging, and describe ways in which Stasys can be customized to implement many existing (and some new) write-ahead-logging variants. Implementations of some of these variants are presented, and benchmarked against popular real-world systems. We conclude with a survey of the technologies the Stasys implementation is based upon.

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

2 Stasys is not a Database

Database research has a long history, including the development of many technologies that our system builds upon. This section explains why databases are fundamentally inappropriate tools for system developers. The problems we present here have been the focus of database systems and research projects for at least 25 years.

2.1 The database abstraction

Database systems are often thought of in terms of the high-level abstractions they present. For instance, relational database systems implement the relational model [10], object oriented databases implement object abstractions, XML databases implement hierarchical datasets, and so on. Before the relational model, navigational databases implemented pointer- and record-based data models.

An early survey of database implementations sought to enumerate the fundamental components used by database system implementors. 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* [2] and *physical database models* [4].

A conceptual mapping might translate a relation into a set of keyed tuples. A physical model would then translate a set of tuples into an on-disk B-Tree, and provide support for iterators and range-based query operations.

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 efficiently support the set of mappings that are built on top of it.

A key observation of this paper is that no known physical data model can support more than a small percentage of today's applications.

Instead of attempting to create such a model after decades of database research has failed to produce one, we opt to provide a transactional storage model that mimics the primitives provided by modern hardware. This makes it easy for system designers to implement most of the data models that the underlying hardware can support, or to abandon the database approach entirely, and forgo the use of a structured physical model or abstract conceptual mappings.

2.2 Extensible transaction systems

This section contains discussion of database systems with goals similar to ours. Although these projects were successful in many respects, they fundamentally aimed to implement a extensible data model, rather than build transactions from the bottom up. In each case, this limits the applicability of their implementations.

2.2.1 Extensible databases

Genesis [3], an early database toolkit, was built in terms of a physical data model and the conceptual mappings described above. It is designed to allow database implementors to easily swap out implementations of the various components defined by its framework. Like subsequent systems (including Stasys), it allows its users to implement custom operations.

Subsequent extensible database work builds upon these foundations. The Exodus [6] database toolkit is the successor to Genesis. It supports the automatic generation of query optimizers and execution engines based upon abstract data type definitions, access methods and cost models provided by its users.

Although further discussion is beyond the scope of this paper, object-oriented database systems and relational databases with support for user-definable abstract data types (such as in Postgres [26]) were the primary competitors to extensible database toolkits. Ideas from all of these systems have been incorporated into the mechanisms that support user-definable types in current database systems.

One can characterize the difference between database toolkits and extensible database servers in terms of early and late binding. With a database toolkit, new types are defined when the database server is compiled. In today's object-relational database systems, new types are defined at runtime. Each approach has its advantages. However, both types of systems aim to extend a high-level data model with new abstract data types, and thus are quite limited in the range of new applications they support. In hindsight, it is not surprising that this kind of extensibility has had little impact on the range of applications we listed above.

2.2.2 Berkeley DB

Berkeley DB is a highly successful alternative to conventional databases. At its core, it provides the physical database (relational storage system) of a conventional database server. In particular, it provides fully transactional (ACID) operations over B-Trees, hashtables, and other access methods. It provides flags that let its users tweak various aspects of the performance of these primitives, and selectively disable the features it provides [24].

With the exception of the benchmark designed to fairly compare the two systems, none of the Stasys applications presented in Section 4 are efficiently supported by Berkeley DB. This is a result of Berkeley DB's assumptions regarding workloads and decisions regarding low level data representation. Thus, although Berkeley DB could be built on top of Stasys, Berkeley DB's data model, and write-ahead-logging system are too specialized to support Stasys.

2.2.3 Better databases

The database community is also aware of this gap. A recent survey [9] enumerates problems that plague users of state-of-the-art database systems, and finds that database implementations fail to support the needs of modern applications. Essentially, it argues that modern databases are too complex to be implemented (or understood) as a monolithic entity.

It supports this argument with real-world evidence that suggests database servers are too unpredictable and unmanagable to scale up the size of today's systems. Similarly, they are a poor fit for small devices. SQL's declarative interface only complicates the situation.

The study concludes by suggesting the adoption of *RISC* database architectures, both as a resource for researchers and as a real-world database system.

RISC databases have many elements in common with database toolkits. However, they take the database toolkit idea one step further, and suggest standardizing the interfaces of the toolkit's internal components, allowing multiple organizations to compete to improve each module. The idea is to produce a research platform that enables specialization and shares the effort required to build a full database [9].

We agree with the motivations behind RISC databases, and to build databases from interchangeable modules exists. In fact, is our hope that our system will mature to the point where it can support a competitive relational database. However this is not our primary goal. Instead of building a modular database, we seek to build a system that enables a wider range of data management options.

3 Transactional Pages

Section 2 described the ways in which a top-down data model limits the generality and flexibility of databases. In this section, we cover the basic bottom-up approach of Stasys: *transactional pages*. Although similar to the underlying write-ahead-logging approaches of databases, particularly ARIES [20], Stasys' bottom-up approach yields unexpected flexibility.

Transactional pages provide the properties of transactions, but only allow updates within a single page in the simplest case. After covering the single-page case, we explore multi-page transactions, which enable a complete transaction system.

In this model, pages are the in-memory representation of disk blocks and thus must be the same size. Pages are a convenient abstraction because the write back of a page (disk block) is normally atomic, giving us a foundation for larger atomic actions. In practice, disk blocks are not always atomic, but the disk can detect partial writes via checksums. Thus, we actually depend only on detection of non-atomicity, which we treat as media failure. One nice property of Stasys is that we can roll forward an individual page from an archive copy to recover from media failures.

A subtlety of transactional pages is that they technically only provide the "atomicity" and "durability" of ACID transactions.[1] This is because "isolation" comes

typically from locking, which is a higher (but compatible) layer. "Consistency" is less well defined but comes in part from transactional pages (from mutexes to avoid race conditions), and in part from higher layers (e.g. unique key requirements). To support these, Stasys distinguishes between *latches* and *locks*. A latch corresponds to an OS mutex, and is held for a short period of time. All of Stasys' default data structures use latches in a way that avoids deadlock. This allows multithreaded code to treat Stasys as a conventional reentrant data structure library. Applications that want conventional isolation (serializability) can make use of a lock manager.

3.1 Single-page Transactions

In this section we show how to implement single-page transactions. This is not at all novel, and is in fact based on ARIES [20], but it forms important background. We also gloss over many important and well-known optimizations that Stasys exploits, such as group commit.These aspects of recovery algorithms are described in the literature, and in any good textbook that describes database implementations. They are not particularly important to our discussion, so we do not cover them.

The trivial way to achieve single-page transactions is simply to apply all the updates to the page and then write it out on commit. The page must be pinned until the transaction commits to avoid "dirty" data (uncommitted data on disk), but no logging is required. As disk block writes are atomic, this ensures that we provide the "A" and "D" of ACID.

This approach scales poorly to multiple pages since we must *force* pages to disk on commit and wait for a (random access) synchronous write to complete. By using a write-ahead log, we can support *no force* transactions: we write (sequential) "redo" information to the log on commit, and then can write the pages later. If we crash, we can use the log to redo the lost updates during recovery.

For this to work, recovery must be able to decide which updates to re-apply. This is solved by using a per-page sequence number called a *log sequence number*. Each log entry contains the sequence number, and each page contains the sequence number of the last applied update. Thus on recovery, we load a page, look at its sequence number, and re-apply all later updates. Similarly, to restore a page from archive we use the same process, but with likely many more updates to apply.

We also need to make sure that only the results of committed transactions still exist after recovery. This is best done by writing a commit record to the log during the commit. If the system pins uncommitted dirty pages in memory, recovery does not need to worry about undoing any updates. Therefore recovery simply plays back unapplied redo records from transactions that have commit records.

However, pinning the pages of active transactions in memory is problematic. First, a single transaction may need more pages than can be pinned at one time. Second, under concurrent transactions, a given page may be pinned forever as long as it has at least one active transaction in progress all the time. To avoid these problems, transaction systems support *steal*, which means that pages can be written back before a transaction commits.

Thus, on recovery a page may contain data that never committed and the corresponding updates must be rolled back. To enable this, "undo" log entries for uncommitted updates must be on disk before the page can be stolen (written back). On recovery, the LSN on the page reveals which UNDO entries to apply to roll back the page. We use the absence of commit records to figure out which transactions to roll back.

Thus, the single-page transactions of Stasys work as follows. An *operation* consists of both a redo and an undo function, both of which take one argument. An update is always the redo function applied to the page (there is no "do" function), and it always ensures that the redo log entry (with its LSN and argument) reaches the disk before commit. Similarly, an undo log entry, with its LSN and argument, always reaches the disk before a page is stolen. ARIES works essentially the same way, but hard-codes recommended page formats and index structures [21].

To manually abort a transaction, Stasys could either reload the page from disk and roll it forward to reflect committed transactions (this would imply "no steal"), or it could roll back the page using the undo entries applied in reverse LSN order. (It currently does the latter.)

This section very briefly described how a simplified write-ahead-logging algorithm might work, and glossed over many details. Like ARIES, Stasys actually implements recovery in three phases: Analysis, Redo and Undo.

3.2 Multi-page transactions

Of course, in practice, we wish to support transactions that span more than one page. Given a no-force/steal single-page transaction, this is relatively easy.

First, we need to ensure that all log entries have a transaction ID (XID) so that we can tell that updates to different pages are part of the same transaction (we need this in the single page case as well). Given single-page recovery, we can just apply it to all of the pages touched by a transaction to recover a multi-page transaction. This works because steal and no-force already

imply that pages can be written back early or late (respectively), so there is no need to write a group of pages back atomically. In fact, we need only ensure that redo entries for all pages reach the disk before the commit record (and before commit returns).

3.3 Nested top actions

So far, we have glossed over the behavior of our system when concurrent transactions modify the same data structure. To understand the problems that arise in this case, consider what would happen if one transaction, A, rearranged the layout of a data structure. Next, assume a second transaction, B, modified that structure, and then A aborted. When A rolls back, its UNDO entries will undo the rearrangement 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 simply 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. Releasing the lock after the modification, but before the end of the transaction, increases concurrency. However, it means that follow-on transactions that use that data may need to abort if a current transaction aborts (*cascading aborts*).

Unfortunately, the long locks held by total isolation cause bottlenecks when applied to key data structures. Nested top actions are essentially mini-transactions that can commit even if their containing transaction aborts; thus follow-on transactions can use the data structure without fear of cascading aborts.

The key idea is to distinguish between the logical operations of a data structure, such as inserting a key, and the physical operations such as splitting tree nodes or or rebalancing a tree. The physical operations do not need to be undone if the containing logical operation (insert) aborts.

Because nested top actions are easy to use and do not lead to deadlock, we wrote a simple Stasys extension that implements nested top actions. The extension may be used as follows:

1. Wrap a mutex around each operation. With care, it may be possible to use finer-grained locks, but it is rarely necessary.

2. Define a *logical* UNDO for each operation (rather than just using a set of page-level UNDO's). For example, this is easy for a hashtable: the UNDO for *insert* is *remove*.

3. For mutating operations, (not read-only), add a "begin nested top action" right after the mutex acquisition, and a "commit nested top action" right before the mutex is released.

If the transaction that encloses the operation aborts, the logical undo will *compensate* for its effects, leaving the structural changes intact.

We have found that it is easy to protect operations that make structural changes to data structures with this recipe. Therefore, we use them throughout our default data structure implementations, although Stasys does not preclude the use of more complex schemes that lead to higher concurrency.

3.4 Blind Writes

As described above, and in all database implementations of which we are aware, transactional pages use LSNs on each page. This makes it difficult to map large objects onto multiple pages, as the LSNs break up the object. It is tempting to try to move the LSNs elsewhere, but then they would not be written atomically with their page, which defeats their purpose.

LSNs were introduced to prevent recovery from applying updates more than once. However, by constraining itself to a special type of idempotent redo and undo entries,[2] Stasys can eliminate the LSN on each page.

Consider purely physical logging operations that overwrite a fixed byte range on the page regardless of the page's initial state. We say that such operations perform "blind writes." If all operations that modify a page have this property, then we can remove the LSN field, and have recovery conservatively assume that it is dealing with a version of the page that is at least as old as the one on disk.

To understand why this works, note that the log entries update some subset of the bits on the page. If the log entries do not update a bit, then its value was correct before recovery began, so it must be correct after recovery. Otherwise, we know that recovery will update the bit. Furthermore, after all REDOs, the bit's value will be the last value it contained before the crash, so we know that undo will behave properly.

We call such pages "LSN-free" pages. Although this technique is novel for databases, it resembles the mechanism used by RVM [23]; Stasys generalizes the concept and allows it to co-exist with traditional pages. Furthermore, efficient recovery and log truncation require only minor modifications to our recovery algorithm. In practice, this is implemented by providing a buffer manager callback for LSN free pages. The callback computes a conservative estimate of the page's LSN whenever the page is read from disk. For a less conservative estimate,

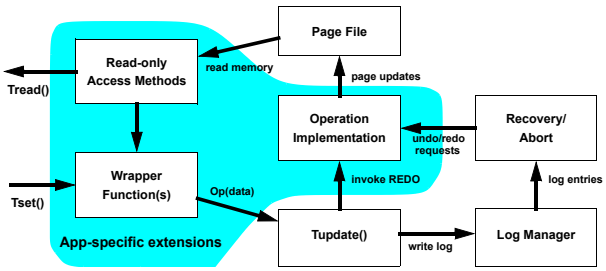

Figure 1: The portions of Stasys that interact with new operations directly.

it suffices to write a page's LSN to the log shortly after the page itself is written out; on recovery the log entry is thus a conservative but close estimate.

Section 4.6 explains how LSN-free pages led us to new approaches for recoverable virtual memory and for large object storage. Section 4.4 uses blind writes to efficiently update records on pages that are manipulated using more general operations.

3.5 Media recovery

Like ARIES, Stasys can recover lost pages in the page file by reinitializing the page to zero, and playing back the entire log. In practice, a system administrator would periodically back up the page file up, thus enabling log truncation and shortening recovery time.

3.6 Summary of Transactional Pages

This section provided an extremely brief overview of transactional pages and write-ahead-logging. Transactional pages are a valuable building block for a wide variety of data management systems, as we show in the next section. Nested top actions and LSN-free pages enable important optimizations. In particular, Stasys allows general custom operations using LSNs, or custom blind-write operations without LSNs. This enables transactional manipulation of large, contiguously stored objects.

4 Extensions

This section describes proof-of-concept extensions to Stasys. Performance figures accompany the extensions that we have implemented. We discuss existing approaches to the systems presented here when appropriate.

4.1 Adding log operations

Stasys allows application developers to easily add new operations to the system. Many of the customizations described below can be implemented using custom log operations. In this section, we describe how to implement an "ARIES style" concurrent, steal/no force operation using full physiological logging and per-page LSN's. Such operations are typical of high-performance commercial database engines.

As we mentioned above, Stasys operations must implement a number of functions. Figure 1 describes the environment that schedules and invokes these functions. The first step in implementing a new set of log interfaces is to decide upon an interface that these log interfaces will export to callers outside of Stasys.

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 undo and redo into a data format of its choosing. This data structure is passed into Tupdate(). Tupdate() copies the data to the log, and then passes the data into the operation's REDO function.

REDO modifies the page file directly (or takes some other action). It is essentially an interpreter for the log entries it is associated with. UNDO works analogously, but is invoked when an operation must be undone (usually due to an aborted transaction, or during recovery).

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

- Pages should only be updated inside REDO and UNDO functions.
- Page updates atomically update the page's LSN by pinning the page.
- If the data seen by a wrapper function must match data seen during REDO, then the wrapper should use a latch to protect against concurrent attempts to update the sensitive data (and against concurrent attempts to allocate log entries that update the data).
- Nested top actions (and logical undo), or "big locks" (total isolation but lower concurrency) should be used to implement multi-page updates. (Section 3.3)

4.2 Experimental setup

We chose Berkeley DB in the following experiments because, among commonly used systems, it provides transactional storage primitives that are most similar to Stasys. Also, Berkeley DB is designed to provide 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 with 1GB of RAM and a 10K RPM SCSI drive formatted using with ReiserFS [22].[3] All results correspond to the mean of multiple runs with a 95% confidence interval with a half-width of 5%.

We used Berkeley DB 4.2.52 as it existed in Debian Linux's testing branch during March of 2005, with the flags DB_TXN_SYNC, and DB_THREAD enabled. These flags were chosen to match Berkeley DB's configuration to Stasys' as closely as possible. In cases where Berkeley DB implements a feature that is not provided by Stasys, we only enable the feature if it improves Berkeley DB's performance.

Optimizations to Berkeley DB that we performed included disabling the lock manager, though we still use "Free Threaded" handles for all tests. This yielded a significant increase in performance because it removed the possibility of transaction deadlock, abort, and repetition. However, disabling the lock manager caused highly 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 multithreaded test in Section 4.3 strictly decreased with increased concurrency. (The other tests were single-threaded.) We also increased Berkeley DB's buffer cache and log buffer sizes to match Stasys' default sizes.

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 that we have produced a reasonably fair comparison. The results presented here have been reproduced on multiple machines and file systems.

4.3 Linear hash table

Although the beginning of this paper describes the limitations of physical database models and relational storage systems in great detail, these systems are the basis of most common transactional storage routines. Therefore, we implement a key-based access method in this section. We argue that obtaining reasonable performance in such a system under Stasys is straightforward. We then compare our simple, straightforward implementation to our hand-tuned version and Berkeley DB's implementation.

The simple hash table uses nested top actions to atomically update its internal structure. It uses a *linear* hash function [16], allowing it to incrementally grow its buffer list. It is based on a number of modular subcomponents. Notably, its bucket list is a growable array of fixed length entries (a linkset, in the terms of the physical database model) and the user's choice of two different linked list

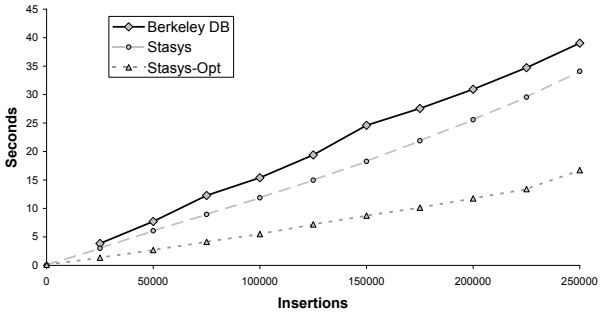

Figure 2: Performance of Stasys and Berkeley DB hashtable implementations. The test is run as a single transaction, minimizing overheads due to synchronous log writes.

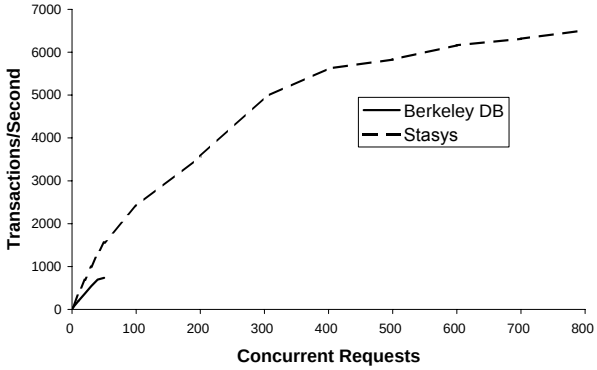

Figure 3: High concurrency performance of Berkeley DB and Stasys. We were unable to get Berkeley DB to work correctly with more than 50 threads. (See text)

implementations.

The hand-tuned hashtable 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 hashtable is a popular, commonly deployed implementation, and serves as a baseline for our experiments.

Both of our hashtables outperform Berkeley DB on a workload that bulk loads the tables by repeatedly inserting (key, value) pairs. The comparison between the Stasys implementations is more enlightening. The performance of the simple hash table shows that straightforward data structure implementations composed from simpler structures can perform as well as the implementations included in existing monolithic systems. The hand-tuned implementation shows that Stasys allows application developers to optimize key primitives.

Figure 3 describes the performance of the two systems under highly concurrent workloads. For this test, we used the simple (unoptimized) hash table, since we

are interested in the performance of a clean, modular data structure that a typical system implementor might produce, not the performance of our own highly tuned, monolithic implementations.

Both Berkeley DB and Stasys can service concurrent calls to commit with a single synchronous I/O.[4] Stasys scaled quite well, delivering over 6000 transactions per second,[5] and provided roughly double Berkeley DB's throughput (up to 50 threads). We do not report the data here, but we implemented a simple load generator that makes use of a fixed pool of threads with a fixed think time. We found that the latency of Berkeley DB and Stasys were similar, showing that Stasys is not simply trading latency for throughput during the concurrency benchmark.

4.4 Object persistence

Numerous schemes are used for object serialization. Support for two different styles of object serialization have been implemented in Stasys. We could have just as easily implemented a persistence mechanism for a statically typed functional programming language, a dynamically typed scripting language, or a particular application, such as an email server. In each case, Stasys' lack of a hard-coded data model would allow us to choose the representation and transactional semantics that make the most sense for the system at hand.

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 serialization library, Oasys. Oasys makes use of pluggable storage modules that implement persistent storage, and includes plugins for Berkeley DB and MySQL.

This section will describe how the Stasys Oasys plugin reduces amount of data written to log, while using half as much system memory as the other two systems.

We present three variants of the Stasys plugin here. The first treats Stasys like Berkeley DB. The second, "update/flush" customizes the behavior of the buffer manager. Instead of maintaining an up-to-date version of each object in the buffer manager or page file, it 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.

By allowing the buffer manager to contain stale data, we reduce the number of times the Stasys Oasys plugin must update serialized objects in the buffer manager. This allows us to drastically decrease the size of the page file. In turn this allows us to increase the size of the application's cache of live objects.

We implemented the Stasys buffer-pool optimization by adding two new operations, update(), which only updates the log, and flush(), which updates the page file.

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. Otherwise, commit would not be durable, unless we queued up log entries, and wrote them all before committing. This would cause Berkeley DB to write data back to the page file, increasing the working set of the program, and increasing disk activity.

Furthermore, objects may be written to disk in an order that differs from the order in which they were updated, violating one of the write-ahead-logging invariants. One way to deal with this is to maintain multiple LSN's per page. This means we would need to register a callback with the recovery routine to process the LSN's (a similar callback will be needed in Section 4.6), and extend Stasys' page format to contain per-record LSN's. Also, we must prevent Stasys' storage allocation routine from overwriting the per-object LSN's of deleted objects that may still be addressed during abort or recovery.

Alternatively, we could arrange for the object pool to cooperate further with the buffer pool by atomically updating the buffer manager's copy of all objects that share a given page, removing the need for multiple LSN's per page, and simplifying storage allocation.

However, the simplest solution, and the one we take here, is based on the observation that updates (not allocations or deletions) of fixed length objects are blind writes. This allows us to do away with per-object LSN's entirely. Allocation and deletion can then be handled as updates to normal LSN containing pages. At recovery time, object updates are executed based on the existence of the object on the page and a conservative estimate of its LSN. (If the page doesn't contain the object during REDO then it must have been written back to disk after the object was deleted. Therefore, we do not need to apply the REDO.) This means that the system can "forget" about objects that were freed by committed transactions, simplifying space reuse tremendously.

The third Stasys plugin, "delta" incorporates the buffer manager optimizations. However, it only writes the changed portions of objects to the log. Because of Stasys' support for custom log entry formats, this optimization is straightforward.

Oasys does not export transactions to its callers. Instead, it is designed to be used in systems that stream objects over an unreliable network connection. Each object update corresponds to an independent message, so there is never any reason to roll back an applied object update. On the other hand, Oasys does support a flush method, which guarantees the durability of updates after it returns. In order to match these semantics as closely as possible, Stasys' update/flush and delta optimizations do

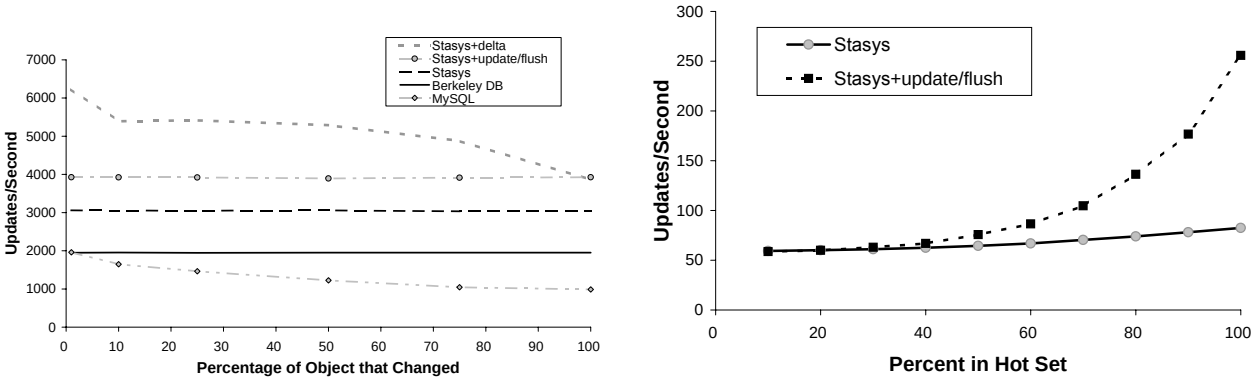

Figure 4: The effect of Stasys object serialization optimizations under low and high memory pressure.

not write any undo information to the log.

These "transactions" are still durable after commit, as commit forces the log to disk. 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 correct durability semantics.)

The operations required for these two optimizations required 150 lines of C code, including whitespace, comments and boilerplate function registrations.[6] Although the reasoning required to ensure the correctness of this code 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 Stasys. We linked the benchmark's executable to the libmysqld daemon library, bypassing the RPC layer. In experiments that used the RPC layer, test completion times were orders of magnitude slower.

Figure 4 presents the performance of the three Stasys optimizations, and the Oasys plugins implemented on top of other systems. As we can see, Stasys performs better than the baseline systems, which is not surprising, since it is not providing the A property of ACID transactions. (Although it is applying each individual operation atomically.)

In non-memory bound systems, the optimizations nearly double Stasys' performance by reducing the CPU overhead of object serialization and the number of log entries written to disk. In the memory bound test, we see that update/flush indeed improves memory utilization.

4.5 Manipulation of logical log entries

Database optimizers operate over relational algebra expressions that correspond to logical operations over

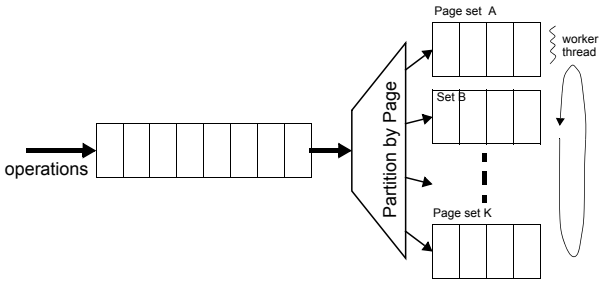

Figure 5: Because pages are independent, we can reorder requests among different pages. Using a log demultiplexer, we partition requests into independent queues, which can be handled in any order, improving locality and merging opportunities.

streams of data. Stasys does not provide query languages, relational algebra, or other such query processing primitives.

However, it does include an extensible logging infrastructure. Furthermore, many operations that make use of physiological logging implicitly implement UNDO (and often REDO) functions that interpret logical requests.

Logical operations often have some nice properties that this section will exploit. Because they can be invoked at arbitrary times in the future, they tend to be independent of the database's physical state. Often, they correspond to operations that programmers understand.

Because of this, application developers can easily determine whether logical operations may be reordered, transformed, or even dropped from the stream of requests that Stasys is processing.

If requests can be partitioned in a natural way, load balancing can be implemented by splitting requests across many nodes. Similarly, a node can easily service streams of requests from multiple nodes by combining them into a single log, and processing the log using operation implementations. For example, this type of opti-

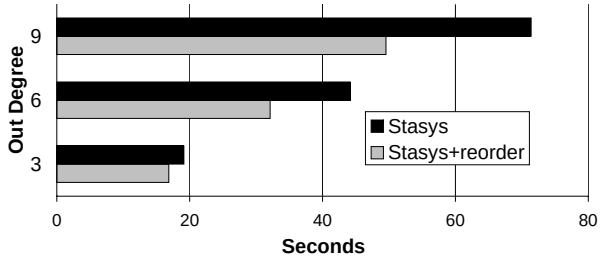

Figure 6: oo7 benchmark style graph traversal. The optimization performs well due to the presence of non-local nodes.

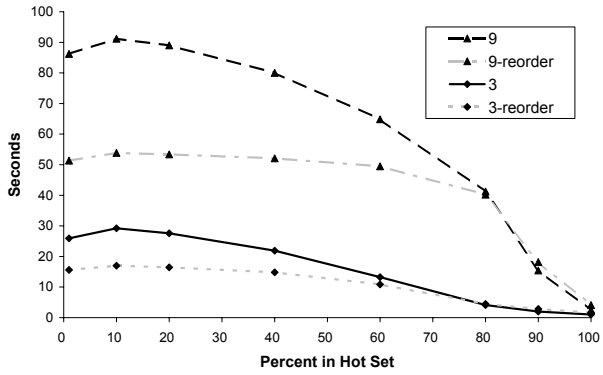

Figure 7: Hot set based graph traversal for random graphs with out-degrees of 3 and 9. Here we see that the multiplexer helps when the graph has poor locality. However, in the cases where depth first search performs well, the reordering is inexpensive.

mization is used by RVM's log-merging operations [23].

Furthermore, application-specific procedures that are analogous to standard relational algebra methods (join, project and select) could be used to efficiently transform the data while it is still layed out sequentially in non-transactional memory.

Although Stasys has rudimentary support for a two-phase commit based cluster hash table, we have not yet implemented networking primitives for logical logs. Therefore, we implemented a single node log-reordering scheme that increases request locality during the traversal of a random graph. The graph traversal system takes a sequence of (read) requests, and partitions them using some function. It then processes each partition in isolation from the others. We considered two partitioning functions. The first divides the page file into equally sized contiguous regions, which increases locality. The second takes the hash of the page's offset in the file, which enables load balancing.

Our benchmarks partition requests by location. We chose the position size so that each partition can fit in Stasys' buffer pool.

We ran two experiments. Both stored a graph of fixed size objects in the growable array implementation that is used as our linear hashtable's bucket list. The first experiment (Figure 6) is loosely based on the oo7 database benchmark. [7]. We hard-code the out-degree of each node, and use a directed graph. OO7 constructs graphs by first connecting nodes together into a ring. It then randomly adds edges between the nodes until the desired out-degree is obtained. This structure ensures graph connectivity. If the nodes are laid out in ring order on disk then it also ensures that one edge from each node has good locality while the others generally have poor locality.

The second experiment explicitly measures the effect of graph locality on our optimization (Figure 7). It extends the idea of a hot set to graph generation. 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 use random edges instead of ring edges for this test. This does not ensure graph connectivity, but we used the same random seeds for the two systems.

When the graph has good locality, a normal depth first search traversal and the prioritized traversal both perform well. The prioritized traversal is slightly slower due to the overhead of extra log manipulation. As locality decreases, the partitioned traversal algorithm's outperforms the naive traversal.

4.6 LSN-Free pages

In Section 3.4, we describe how operations can avoid recording LSN's on the pages they modify. Essentially, operations that make use of purely physical logging need not heed page boundaries, as physiological operations must. Recall that purely physical logging interacts poorly with concurrent transactions that modify the same data structures or pages, so LSN-Free pages are not applicable in all situations.

Consider the retrieval of a large (page spanning) object stored on pages that contain LSN's. The object's data will not be contiguous. Therefore, in order to retrieve the object, the transaction system must load the pages contained on disk into memory, and perform a byte-by-byte copy of the portions of the pages that contain the large object's data into a second buffer.

Compare this approach to a modern filesystem, which allows applications to perform a DMA copy of the data into memory, avoiding the expensive byte-by-byte copy of the data, and allowing the CPU to be used for more productive purposes. Furthermore, modern operating systems allow network services to use DMA and network adaptor hardware to read data from disk, and send it over a network socket 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 could be performed by performing a DMA write to a portion of the log file. However, doing this complicates log truncation, and does not address the problem of updating the page file. We suspect that contributions from the log based filesystem [11] literature can address these problems in a straightforward fashion. In particular, we imagine storing portions of the log (the portion that stores the blob) in the page file, or other addressable storage. In the worst case, the blob would have to be relocated in order to defragment the storage. Assuming the blob was relocated once, this would amount to a total of three, mostly sequential disk operations. (Two writes and one read.) However, in the best case, the blob would only need to written once. In contrast, a conventional atomic blob implementation would always need to write the blob twice.

Alternatively, we could use DMA to overwrite the blob in the page file in a non-atomic fashion, providing filesystem style semantics. (Existing database servers often provide this mode based on the observation that many blobs are static data that does not really need to be updated transactionally. [19]) Of course, Stasys could also support other approaches to blob storage, such as B-Tree layouts that allow arbitrary insertions and deletions in the middle of objects [8].

Finally, RVM, recoverable virtual memory, made use of LSN-free pages so that it could use mmap() to map portions of the page file into application memory[23]. However, without support for logical log entries and nested top actions, it would be difficult to implement a concurrent, durable data structure using RVM. We plan to add RVM style transactional memory to Stasys in a way that is compatible with fully concurrent collections such as hash tables and tree structures.

5 Related Work

This paper has described a number of custom transactional storage extensions, and explained why can Stasys support them. This section will describe existing ideas in the literature that we would like to incorporate into Stasys.

Different large object storage systems provide different API's. Some allow arbitrary insertion and deletion of bytes [8] or pages [19] within the object, while typical filesystems provide append-only storage allocation [18]. Record-oriented file systems are an older, but still-used [13] alternative. Each of these API's addresses different workloads.

While most filesystems attempt to lay out data in logically sequential order, write-optimized filesystems lay files out in the order they were written [11]. Schemes to improve locality between small objects exist as well. Relational databases allow users to specify the order in which tuples will be layed out, and often leave portions of pages unallocated to reduce fragmentation as new records are allocated.

Memory allocation routines also address this problem. For example, the Hoard memory allocator is a highly concurrent version of malloc that makes use of thread context to allocate memory in a way that favors cache locality [5].

Finally, many systems take a hybrid approach to allocation. Examples include databases with blob support, and a number of filesystems [22, 18].

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 [11] and POSTGRES [26] to implement this.

Starburst [17] provides a flexible approach to index management, and database trigger support, as well as hints for small object layout.

The Boxwood system provides a networked, fault-tolerant transactional B-Tree and "Chunk Manager." We believe that Stasys is an interesting complement to such a system, especially given Stasys' focus on intelligence and optimizations within a single node, and Boxwood's focus on multiple node systems. In particular, it would be interesting to explore extensions to the Boxwood approach that make use of Stasys' customizable semantics (Section 4.1), and fully logical logging mechanism. (Section 4.5)

6 Future Work

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

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

The complexity of the core of Stasys is our primary concern, as it contains the hard-coded policies and assumptions. Over time, the core has shrunk as functionality has been 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 between sets of components. Over time, we hope to shrink Stasys' core to the point where it is simply a resource manager and a set of implementations of a few unavoidable algorithms related to write-ahead-logging. For instance, we suspect that support for appropriate callbacks will allow us to hard-code a generic recovery algorithm into the system. Similarly, any code that manages book-keeping information, such as LSN's may be general enough to be hard-coded.

Of course, we also plan to provide Stasys' current functionality, including the algorithms mentioned above as modular, well-tested extensions. Highly specialized Stasys extensions, and other systems would be built by reusing Stasys' default extensions and implementing new ones.

7 Conclusion

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

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

8 Acknowledgements

The idea behind the Oasys buffer manager optimization is from Mike Demmer. He and Bowei Du implemented Oasys. Gilad Arnold and Amir Kamil implemented for pobj. Jim Blomo, Jason Bayer, and Jimmy Kittiyachavalit worked on an early version of Stasys.

Thanks to C. Mohan for pointing out the need for tombstones with per-object LSN's. Jim Gray provided feedback on an earlier version of this paper, and suggested we use a resource manager to manage dependencies within Stasys' API. Joe Hellerstein and Mike Franklin provided us with invaluable feedback.

9 Availability

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

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

References

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

[2] BATORY, D. S. Conceptual-to-internal mappings in commercial database systems. In *Proceedings of the 3rd SIGACT-SIGMOD symposium on Principles of database systems* (1984), pp. 70–78.

[3] 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.

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

[5] 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.

[6] 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 *Proceedings on the 1986 international workshop on Object-oriented database systems* (1986), pp. 52–65.

[7] CAREY, M. J., DEWITT, D. J., AND NAUGHTON, J. F. The oo7 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (1993), pp. 12–21.

[8] 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'86 Twelfth International Conference on Very Large Data Bases* (1986), pp. 91–100.

[9] CHAUDHURI, S., AND WEIKUM, G. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Proceedings of the 26th International Conference on Very Large Databases* (2000).

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

[11] DESIGN, T., AND OF A LOG-STRUCTURED FILE SYSTEM, I. Mendel rosenblum and john k. ousterhout. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (1992).

[12] ENGLER, D. R., AND KAASHOEK, M. F. Exterminate all operating system abstractions. *HotOS* (1995).

[13] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003), pp. 29–43.

[14] Hibernate: Relational persistence for Java and .NET. http://www.hibernate.org/.

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

[16] LITWIN, W. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th Conference on Very Large Dabatases* (1980), pp. 224–232.

[17] LOHMAN, G. M., 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.

[18] 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).

[19] Microsoft SQL Server 2005.

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

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

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

[23] 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 (Februrary 1994), 33–57.

[24] SELTZER, M., AND OLSEN, M. LIBTP: Portable, modular transactions for UNIX. In *Proceedings of the 1992 Usenix Conference* (Janurary 1992).

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

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

[27] ZEEBERG, B., RISS, J., D, D. K., BUSSEY, K., UCHIO, E., LINEHAN, W., BARRET, J., AND WEINSTEIN, J. Mistaken identifiers: Gene name errors can be introduced inadvertently when using Excel in bioinformatics. *BMC Bioinformatics* (2004).

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".

[2]Idempotency does not guarantee that $f(g(x)) = f(g(f(g(x))))$. Therefore, idempotency does not guarantee that it is safe to assume that a page is older than it is.

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

[4]The multi-threaded benchmarks presented here were performed using an ext3 filesystem, as high concurrency caused both Berkeley DB and Stasys to behave unpredictably when ReiserFS was used. However, Stasys' multi-threaded throughput was significantly better that Berkeley DB's under both filesystems.

[5]The concurrency test was run without lock managers, and the transactions obeyed the A, C, and D properties. Since each transaction performed exactly one hashtable write and no reads, they also obeyed I (isolation) in a trivial sense.

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