Failure-Atomic `msync()`: A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data

Stan Park
University of Rochester
park@cs.rochester.edu

Terence Kelly
Hewlett-Packard Laboratories
terence.p.kelly@hp.com

Kai Shen
University of Rochester
kshen@cs.rochester.edu

Abstract

Preserving the integrity of application data across updates is difficult if power outages and system crashes may occur during updates. Existing approaches such as relational databases and transactional key-value stores restrict programming flexibility by mandating narrow data access interfaces. We have designed, implemented, and evaluated an approach that strengthens the semantics of a standard operating system primitive while maintaining conceptual simplicity and supporting highly flexible programming: *Failure-atomic* `msync()` commits changes to a memory-mapped file atomically, even in the presence of failures. Our Linux implementation of failure-atomic `msync()` has preserved application data integrity across hundreds of whole-machine power interruptions and exhibits good microbenchmark performance on both spinning disks and solid-state storage. Failure-atomic `msync()` supports higher layers of fully general programming abstraction, e.g., a persistent heap that easily slips beneath the C++ Standard Template Library. An STL `<map>` built atop failure-atomic `msync()` outperforms several local key-value stores that support transactional updates. We integrated failure-atomic `msync()` into the Kyoto Tycoon key-value server by modifying exactly one line of code; our modified server reduces response times by 26–43% compared to Tycoon's existing transaction support while providing the same data integrity guarantees. Compared to a Tycoon server setup that makes almost no I/O (and therefore provides no support for data durability and integrity over failures), failure-atomic `msync()` incurs a three-fold response time increase on a fast Flash-based SSD—an acceptable cost of data reliability for many.

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

1. Introduction

Preserving the integrity of application data is the paramount responsibility of computing systems, and sudden system crashes due to power outages and kernel panics pose the most serious threats to application data integrity. At a minimum, applications must be able to recover a consistent application state from durable storage following such failures.

Networked data processing—increasingly the norm—often imposes the added requirement that an application *synchronously* commit a consistent state to durable media before releasing outputs. Consider, for example, a banking server handling a request to transfer money from a savings account to a checking account. The server must synchronously commit this transaction to storage before informing the client that the transfer has completed, lest a server crash effectively erase the transfer and the client's checks bounce. More general distributed computing scenarios impose more sophisticated requirements on global consistency [14, 21]; as in simple client-server interactions, the need to commit data synchronously is sometimes unavoidable. Indeed, the motivation for our work is to provide efficient kernel support for the Ken distributed rollback-recovery protocol [25], currently available as a portable open-source userspace library [24] that provides fault tolerance for several independent distributed systems toolkits [19, 22, 26, 45].

Existing support for preserving application data integrity in the face of crashes leaves much to be desired. Most operating systems and file systems provide only *ordering* primitives which by themselves do not guarantee *atomic and consistent* application data updates in the face of sudden crashes. POSIX `msync()`, for example, can leave file data inconsistent if a crash occurs after a subset of dirty pages have been eagerly written back to the file or if a crash occurs during a call to `msync()`. Programmers have the option of writing application-specific reliability mechanisms atop existing OS primitives, but manually inserting checkpoints is tedious and homebrew crash-recovery code is notoriously buggy.

Programmers therefore frequently rely upon relational databases or key-value stores to protect application data from crashes. While reliable, such mechanisms are not al-

ways well aligned with the style of programming that the developer would prefer if reliability were not a requirement. Imposing the relational model of conventional databases on an application's data can be awkward, and the narrow `put()/get()` interface of key-value stores is seldom an ergonomic substitute for the unrestricted manipulation of in-memory data that programmers ordinarily enjoy.

We postulate that in many cases what programmers really want is the convenience and generality of ordinary main-memory data structures and algorithms coupled with a simple lightweight mechanism to atomically commit data to storage at moments when the programmer knows that the data are consistent. A good solution would enable programmers to evolve application data on durable media from one consistent state to the next without fear that sudden crashes may destroy or corrupt data and without cumbersome programming.

Our solution, failure-atomic `msync()`, is a conceptually simple extension to the familiar POSIX `mmap()/msync()` interfaces. A file mapped into memory with the new flag `mmap(MAP_ATOMIC)` is guaranteed to be in the state it was in *before* it was mapped into memory, or the state it was in *after* the most recent successful `msync()` call, even if system crashes occur. In our tests, failure-atomic `msync()` has preserved application data integrity across hundreds of whole-machine power disconnections without losing or corrupting a single byte of application data.

By itself, failure-atomic `msync()` directly addresses a demand that programmers have explicitly voiced [29]. It also can serve as the foundation for higher layers of programming abstraction that inherit the strong data integrity guarantees of failure-atomic `msync()`. For example, we found it easy to build atop failure-atomic `msync()` a persistent heap that in turn can replace the default C++ Standard Template Library (STL) memory allocator. The net result is a powerful and efficient combination of durable data consistency with STL's comprehensive collection of data structures and algorithms. Finally, a backward-compatible interface makes it remarkably easy to integrate failure-atomic `msync()` into large and complex legacy applications. For example, by modifying *one line of code* we enabled the Kyoto Tycoon key-value server to take full advantage of failure-atomic `msync()`, which significantly outperformed Tycoon's native transaction support while providing the same data integrity guarantees.

The remainder of this paper is organized as follows: We begin with a review of related work in Section 2. Section 3 describes the interface, semantics, and system support of failure-atomic `msync()`. Section 4 explains how failure-atomic `msync()` facilitates both the development of new applications and the retrofitting of enhanced fault tolerance onto legacy applications. Section 5 presents a series of experiments that evaluate the robustness and performance of our implementation of failure-atomic `msync()`, and Section 6 concludes.

2. Related Work

Transactional databases support data durability and consistency through relational SQL [42] or object-oriented interfaces [18]. NoSQL key-value stores like Kyoto Cabinet [15] and LevelDB [12] also preserve data integrity over system failures. These systems mandate specific data access interfaces that restrict programming flexibility. In comparison, our failure-atomic `msync()` has a small, backward-compatible addition to the POSIX I/O interface such that it affords easy adoption and full flexibility to the programmers. Furthermore, it requires minimal porting effort for use by existing applications that desire durability and integrity for their data in memory.

Existing file system atomic I/O techniques include journaling, shadowing, and soft updates. In journaling, an atomic I/O operation is recorded in a REDO log before writing to the file system. A failure after a partial write can be recovered at system restart by scanning and committing the REDO log. In shadowing [1, 23], writes to existing files are handled in a copy-on-write fashion to temporary shadow blocks. The final commit is realized through one atomic I/O write to a file index block that points to updated shadow data/index blocks. While the traditional shadowing requires "bubbling up to the root" of the file system index structure, short-circuit shadowing [10] allows a direct change of an intermediate block to save work at the cost of disrupting sequential writes. Finally, soft updates [17] carefully order the block writes in file system operations such that any mid-operation failure will always leave the file system metadata in a consistent state (except for possible space leaking on temporarily written blocks). However, the existing file system atomic I/O support [35] is primarily concerned with protecting *file system* data structure consistency rather than *application* data integrity. In particular, the ext4 file system "data" mode journaling cannot preserve application-level data integrity semantics about which it currently has no way to know.

For data consistency in a networked system (e.g., Web clients/servers), it is furthermore important to commit data to durable storage before emitting dependent output to the outside world. Output-triggered commits are supported in "Rethink the Sync" [31], which contains OS-level techniques to track dependencies between dirty file buffers and kernel objects, enabling automatically committing dependent dirty file buffers before any output message is released. We take a different approach of providing a simple, intuitive programming interface for direct application control. Our approach is less automated but more flexible. Adding synchrony to failure-atomic I/O may incur substantial overhead [39]. The design and implementation of failure-atomic `msync()` attempt to support durable, consistent data preservation at high efficiency.

Atomic I/O transactions have been supported in both file systems and in storage. Generally speaking, application-level data integrity semantics are not visible at the storage firmware and therefore storage-level transactions [36] are not suitable for protecting application data integrity. At the operating system level, Stasis supports transactional I/O on memory pages through UNDO logging [38]. TxOS [34] connects its transactional memory support with file system journaling to enable atomic storage operations. Compared with I/O transactions, our failure-atomic `msync()` presents a simple, POSIX-compatible programming interface that is less complex and easier to use. It is worth noting that Microsoft Windows Vista introduced an atomic file transaction mechanism (TxF) but the vendor deprecates and may discontinue this feature, noting "*extremely limited developer interest ... due to its complexity and various nuances*" [28]. Some transactional I/O systems [34, 38] enable atomic I/O over failures as well as concurrent accesses. Failure-atomic `msync()` focuses on failure-atomicity while leaving concurrency management to the applications (through mutex locks or other synchronization means).

Rio Vista [27] was an early effort that supports data consistency over operating system failures on persistent memory but did not support data consistency over power failures. RVM [37] is similar in spirit to failure-atomic `msync()`, though utilizing a different interface and focusing on virtual memory support rather than mapped files. With continuing advances in non-volatile memory (NVRAM) hardware technologies [8, 11], recent studies have proposed a new NVRAM-based file system design [10], new data access primitives (including Mnemosyne [44], NV-heaps [9], and CDDS [43]), as well as fast failure recovery [30]. Unfortunately, today's NVRAM manufacturing technologies still suffer from low space density (or high $/GB) and stability/durability problems. Until these problems are resolved, today's storage hardware (mechanical disks and NAND Flash-based solid-state drives) and system software (block-based file systems) are likely to remain. To realize our primary objectives of ease-of-use and fast adoption, failure-atomic `msync()` targets the software/hardware stacks running in today's systems.

Supporting a persistent heap between volatile memory and durable storage is a classic topic. Atkinson et al. proposed PS-algol, a database programming model that allows programmers to directly manipulate data structures on a heap [2] while an underlying system properly and promptly moves data from the heap to persistent storage [3]. O'Toole et al. presented a replicating garbage collector that cooperates with a transaction manager to provide durable, consistent storage management [32]. Guerra et al. identify a consistent data version in the heap through pointer chasing from a root data unit and atomically commit each data version [20]. At a lower level of abstraction, our failure-atomic `msync()` can easily implement a persistent heap with data integrity and high efficiency but it also allows other programming paradigms on memory-mapped data.

The belief that programmers benefit from the convenience of manipulating durable data via conventional main-memory data structures and algorithms dates back to MULTICS [4], which inspired today's memory-mapped file interfaces. Failure-atomic `msync()` retains the ergonomic benefits of memory-mapped files and couples them with strong new data-integrity guarantees.

Finally, our work is related to data center state management systems such as Bigtable [7] and Dynamo [13] but with different emphases. While centrally managed data centers can impose a unified data access model and distributed coordination, failure-atomic `msync()` enables small local adjustment of existing operating system support at individual hosts, which is more suitable for the vast majority of independent application development scenarios.

3. Interface and System Support

Failure-atomic `msync()` is a simple OS-supported mechanism that allows the application programmer to evolve durable application data atomically, in spite of failures such as fail-stop kernel panics and power outages. Failure-atomic `msync()` guarantees that a memory-mapped file will always either be in the state it was in immediately after the most recent `msync()` (or the state it was in at the time of `mmap()` if `msync()` has not been called).

Because its semantics lie at the high-level interface between the operating system and applications, failure-atomic `msync()` does not fundamentally depend on particular durable media (whether block device or not)—today's hard disks and SSDs and forthcoming non-volatile memory are compatible. Indeed, failure-atomic `msync()` seems to be an ideal interface to novel mechanisms for taking memory checkpoints almost *instantaneously* by versioning multilevel-cell NVRAM [46].

In addition to having flexibility in the underlying storage device, the concept of failure-atomic `msync()` allows multiple implementations. Journaling, shadow copy, and soft updates are all viable techniques that allow consistent updates to a file system. In this paper, we describe our journaling-based system support.

3.1 Interface and Semantics

The interface to failure-atomic `msync()` is simply the familiar `mmap()` and `msync()` system calls. In order to enable failure-atomic `msync()`, the programmer merely needs to specify a new `MAP_ATOMIC` flag to `mmap()` in addition to any other flags needed. The programmer can access the memory-mapped region in the customary fashion. When the application state is deemed consistent by the programmer, `msync(MS_SYNC)` is called.

Two POSIX-standardized `msync()` flags—which are currently no-ops in Linux—illustrate the fundamental har-

mony between the conventional, standardized `msync()` concept and our failure-atomicity extension.

- First, consider `msync(MS_ASYNC)`, which on Linux currently results in a no-op. POSIX allows this call to "*return immediately once all the write operations are initiated or queued for servicing*." The composition of failure atomicity with `MS_ASYNC` has clean and simple semantics—"*define and enqueue an atomic batch of updates to the backing file, but don't wait for the updates to be applied to durable media*."
- Next, consider the `MS_INVALIDATE` flag, which like `MS_ASYNC` currently results in a no-op on Linux. The composition of failure atomicity with `MS_INVALIDATE` leads to straightforward and meaningful semantics: `msync(MS_INVALIDATE)` on a `mmap(MAP_ATOMIC)` file would abort and roll back changes made to the in-memory mapping, leaving the backing file unchanged.

Implementing standard POSIX `MS_ASYNC` and `MS_INVALIDATE` semantics in Linux is beyond the scope of our work, which focuses on the synchronous semantics required by networked applications that must commit state prior to releasing messages. We simply note that the concept of failure atomicity introduces no semantic difficulties in composition with the existing standardized interface.

Isolation, data-race prevention, or other multi-threaded or multi-process concurrency control is not a goal of failure-atomic `msync()`. Failure-atomicity adds no new complications to concurrency, leaving the programmer's responsibilities unchanged. We further discuss this issue in Section 3.3.

3.2 Kernel and File System Support

Design Overview Implementing failure-atomic `msync()` entails two fundamental requirements that differ from the standard behavior of `msync()`: First, we must ensure that the memory-mapped file is altered only in response to explicit `msync()` calls. For example, we must ensure that modified pages of the in-memory file image are *not* written back to the underlying file for other reasons, e.g., in response to memory pressure or as a consequence of a timer-based periodic writeback mechanism. Second, we must ensure that when `msync()` is called, the set of dirty in-memory pages must be committed in a failure-atomic fashion; following a sudden crash *during* the `msync()` call, the file must remain in the state it was in before the call.

Preventing premature writeback of dirty pages consists mainly in identifying locations in the kernel where such writebacks may occur and disabling them in a safe way. Making `msync()` calls failure-atomic is simplified—at least in principle—if we leverage the failure-atomicity properties already present in an underlying journaling mechanism. Once an atomic set of updates is safely written to the durable journal, they are guaranteed to survive a failure. Implementing failure-atomic `msync()` is conceptually simple, e.g., modifying memory management and leveraging the failure-atomicity properties of a journaling mechanism. In practice, of course, the devil is in the details, which we describe at length below. Overall, we found that extending a complex and mature OS (Linux and ext4 journaling file system) to support failure-atomic `msync()` was a manageable exercise, and we believe that similar modifications could be made without prohibitive effort to other Unix-like operating systems.

Implementation Failure-atomic `msync()` involves two logical goals. First, a mapped file must be maintained in a consistent state until the programmer defines the next consistent state by calling `msync()`. That is, the file must be protected from non-explicit modification. To this end, we disable any writeback of pages of an atomically mapped region unless the writeback was initiated by an explicit `sync`-type call. In order to identify such protected pages, we set a new page flag `Atomic` when a page in an atomically mapped region is first accessed.

Second, evolving file state must occur in such a way that failure during evolution does not compromise the consistency of the file data. In our journaling-based approach, we leverage JBD2, the journaling layer of the ext4 file system, to achieve failure-atomic writes to durable storage. While JBD2 is typically used to journal file system metadata in order to maintain file system consistency, it is also capable of performing data journaling, i.e., journaling of file data. Data journaling allows dirty data to be written to durable storage in an out-of-place fashion, i.e., to a secondary location in lieu of the original data location, which is necessary for maintaining file data consistency in the case of failures during the writeback of a set of updates.

The standard data journaling, however, provides no interface for applications to specify units of atomic I/O for consistency. In our implementation, failure-atomic `msync()` collects the set of dirty pages at an `msync()` call and encapsulates them into a single handle—the unit of failure-atomic I/O guaranteed by the journaling JBD2 layer. JBD2 is capable of providing atomic transaction guarantees via a checksummed journal entry, so failures during an atomic `msync()` will merely result in an incomplete journal entry which will be discarded. Once the journal entry is safely on durable storage, the actual file data can be written to the file system in the future.

While implementing failure-atomic `msync()` on ext4, we were initially tempted to use its existing I/O functions like `ext4_writepages()` and `ext4_writepage()`. Unfortunately `ext4_writepages()` is intended for a contiguous range of the address space; for non-contiguous range operations, `ext4_writepage()` is called repeatedly for each dirty page in the range. When ext4 is mounted in data journaling mode, repeatedly calling `ext4_writepage()` causes each page to be journaled in a separate handle. JBD2 only guarantees each handle to be performed atomically so it is

possible that, during this per-page journaling process, some pages are committed to the durable storage before others. A failure after the commit of at least one but before the full page set (within an `msync()`) commits creates a window of vulnerability in which inconsistency may occur. Therefore we had to adopt a new implementation path without using these existing ext4 I/O functions.

Journaled Writeback Consider a file system with REDO log-based journaling (such as Linux ext4/JBD2). Once the journal entry for the new pages has been safely written, the data is effectively made durable even though the in-place file content has not yet been updated—if the system fails at that moment, journal recovery will ensure that the file content will be consistent and up-to-date. Therefore, the eventual writeback of journaled pages can be arbitrarily delayed without violating data durability semantics. Note the distinction between two kinds of dirty pages—*file system-layer* volatile dirty pages that would be lost at system failures and *journaled* dirty pages that are already made effectively durable by journal REDO logging. We distinguish the writeback of these two kinds of dirty pages as *file system-layer* writeback and *journaled* writeback, respectively. We illustrate the two kinds of dirty pages and their writeback in Figure 1.

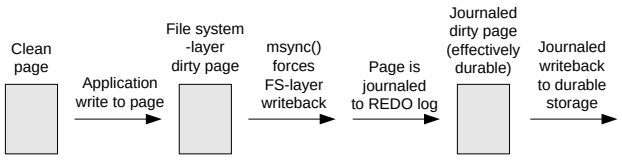


Figure 1: Lifetime of a page and two kinds of writeback.

The general operating system design has not given particular attention to the management of journaled writeback and some (like Linux) manage it indifferently from the conventional file system-layer writeback. Such oblivion can lead to substantial performance costs for applications that make frequent data I/O to the durable storage (including those that adopt failure-atomic `msync()`). Particularly in Linux, when `msync()` is called repeatedly on a memory-mapped file, the processing of an `msync()` call does not distinguish between file system-layer dirty pages and journaled dirty pages. Consequently, all dirty pages are passed to the file system for writeback and file systems like ext4, when seeing a writeback request on a previously journaled page, will commit the pages to storage. This is unnecessary for journaled pages that are already effectively durable. We call such an approach *eager journaled writeback*.

Eager journaled writeback introduces additional work in the synchronous context, which leads to additional application wait time. We explore asynchronous writeback of already journaled pages. In particular, we distinguish between file system-layer dirty pages and journaled dirty pages. At `msync()`, we immediately writeback file system-layer dirty pages but leave the journaled dirty pages for a possible delay. While the operating system generally contains a mechanism for asynchronous writeback, it may not be appropriate for a failure-atomic `msync()`-intensive system that performs frequent I/O to preserve data durability and integrity. In particular, there is a cost associated with the accumulation of journaled dirty pages in a memory-mapped file. Specifically, all dirty pages will be scanned at an `msync()` and distinguishing between many dirty pages (file system-layer or journaled) may incur high page lookup costs (and associated TLB misses). We limit such costs by placing an upper bound on the accumulation of journaled dirty pages within a file before the asynchronous writeback of those dirty pages is initiated. We use 128 pages as the threshold in our experiments.

While asynchronous journaled writeback generally outperforms the eager journaled writeback due to less work in the synchronous context, it may be less effective for faster I/O devices. On extremely fast devices, the additional latency of performing non-critical I/O in the synchronous context may be outweighed by the costs of additional page lookups and higher memory pressure. We evaluate both approaches in Section 5.

3.3 Additional Issues

Preventing asynchronous file system-layer writeback is necessary to maintain file consistency under failure-atomic `msync()`, but it effectively pins file system-layer dirty pages in memory and thus constrains the operating system's responses to memory pressure. Fortunately an OS may reconcile the requirements of failure-atomic `msync()` with memory pressure using a straightforward mechanism. Specifically, when memory pressure arises, the OS may write dirty pages in `MAP_ATOMIC` regions to temporary durable storage (e.g., swap space) rather than to the backing file; when failure-atomic `msync()` is called, these pages can be swapped back in. A more complex solution, which minimizes expensive I/O, is to directly relink the blocks on storage. We have not implemented this mechanism in our current prototype.

We must be careful to distinguish between two conceptually distinct concerns that are easily conflated, in part because terms such as "transactional" and "atomicity" are widely used to describe them individually or in combination: (1) integrity-preserving update of durable data in the face of severe failures, e.g., power outages; and (2) orderly concurrent access to shared (but not necessarily durable) data, e.g., to prevent data races in multi-threaded software. For brevity we refer to these concerns as *consistent update* and *concurrency isolation*, respectively. Over the past decade the rise of multicore processors inspired intense research interest in improved concurrency isolation mechanisms such as transactional memory. More recent research has enabled transactional storage [38] and transactional operating systems [34]

to provide *unified* support for consistent update on durable media and concurrency isolation.

Failure-atomic `msync()` conforms to the traditional POSIX approach of *separating* consistent update from concurrency isolation, which has long been viewed as positively desirable. For example, Black argued that in the operating system context, "*[transaction support] should be subdivided into independent components, and that each component be isolated from its implementation by a well defined interface*" [6]. Failure-atomic `msync()` contributes a well-defined new consistent update mechanism by strengthening the well-known semantics of `msync()`. The result remains compatible with concurrency isolation mechanisms such as mutexes. In a multi-threaded program that employs failure-atomic `msync()` on shared buffer, a simple expedient suffices to prevent concurrency from wreaking havoc during `msync()` calls: An ordinary reader-writer lock confers upon threads that hold a read lock the privilege of modifying the buffer, and grants to a thread holding a write lock the privilege of calling `msync()` on the buffer.

While separated consistent update and concurrency isolation support provides flexibility, we acknowledge, however, that it does not always produce the best performance or programmability. Compared to an ideal transactional system that supports both consistent update and concurrency isolation while allowing the maximum level of concurrency among threads, failure-atomic `msync()`-based support falls short. In particular, while failure-atomic `msync()` can combine with coarse-grained mutexes, it would restrict the concurrency. Alternatively, failure-atomic `msync()` can combine with fine-grained mutexes to enable concurrency but fine-grained mutexes are hard to program. Finally, we point out that our failure-atomicity system support (though not our atomic `msync()` API) may integrate into a transactional memory system to achieve consistent update and concurrency isolation. Such an effort requires further investigation that falls beyond the scope of this paper.

4. Application Development

Failure-atomic `msync()` facilitates the development of new software and also the enhancement of existing software. Section 4.1 describes how to layer software abstractions atop our new primitive in such a way that the higher layers inherit its benefits. Section 4.2 illustrates how the backward-compatibility of failure-atomic `msync()` makes it easy to strengthen the crash resilience of complex legacy applications. Performance evaluations for both scenarios are presented respectively in Section 5.3 and Section 5.4.

4.1 Persistent Heap and C++ STL

The widely-anticipated advent of non-volatile memory has sparked interest in persistent heaps as an NVRAM interface [9, 44]. The attractions of persistent heaps as a *programming abstraction*, however, are independent of the underlying durability technology, and recent contributions include persistent heaps designed to work with conventional storage [20]. Failure-atomic `msync()` provides a convenient foundation for persistent heaps. Here we sketch a minimalist design that illustrates the basic concepts, works rather well in practice, and integrates easily with rich general-purpose programming libraries.

A large but initially sparse *heap file* provides storage-backed memory. Metadata contained in the heap file includes the virtual memory address at which the file should be mapped, free-list management metadata, and a pointer to the root of user-defined data structures contained in the heap. An initialization function uses failure-atomic `msync()` to map the heap file into a process's address space at the fixed location specified in the file itself; the same `init()` function handles both the very first initialization of a heap file and recovery of the heap from the heap file following a crash or an orderly process shutdown. Address Space Layout Randomization (ASLR) could in principle thwart our attempt to map the heap file at a fixed address; in our experience such problems do not occur, and ASLR can be disabled as a last resort. A memory allocator that exposes a `malloc()/free()` interface manages the memory pool and free list backed by the heap file. A `set()/get()` interface allows access to the heap's root pointer, and a `sync()` function allows the user to atomically synchronize the in-memory heap with the heap file; under the hood, the `sync()` function simply invokes our failure-atomic `msync()` on the entire heap mapping. Our persistent heap implementation consists of under 200 lines of C code.

Programming with a persistent heap based on failure-atomic `msync()` differs little from ordinary in-memory programming, except for a handful of commonsense restrictions that also apply to alternative persistence strategies. Programs must be able to reach data structures in the persistent heap from the root pointer, e.g., after recovery from a crash; a convenient way to do this is to use the root as an entry to a hash table that maps data structure names to corresponding pointers. Programs should not store in the persistent heap pointers to data outside the heap. Programs may `sync()` the heap whenever it is consistent; no threads may be writing to the heap during a `sync()` call, just as no threads may write to a buffer as it is being processed by a `write()` call. Finally, just as the conventional heap is private to a conventional process, the heap file of a persistent heap based on failure-atomic `msync()` should not be used for sharing data between concurrent processes.

We believe that the `sync()` interface to our persistent heap is more convenient and less error-prone than a "begin/end transaction" interface. Experience has shown that correctly pairing begin and end calls (or lock/unlock, or open/close, etc.) is fertile ground for bugs. The semantics of `sync()` is simple and requires no pairing. A `sync()` call simply means, "the heap is in a consistent state."

The C++ Standard Template Library (STL) is designed to make it easy for its default memory allocator to be replaced by a user-defined allocator. It takes roughly a dozen lines of straightforward code to insert our persistent heap's `malloc()` and `free()` into STL in such a way that both programmer-visible `new` and `delete` calls and internal allocations employ our functions. The result is that nearly the full power of STL is coupled to nearly effortless persistence. The programmer remains responsible for calling `sync()` and for finding data structures in the persistent heap from the heap's root. Programs that employ our persistent heap beneath STL must prevent constructors that allocate memory from being called before the persistent heap `init()` function, keeping in mind that C++ static-variable constructors can be called in an uncontrollable order before a program's `main()` function is called. Avoiding static-variable constructors that allocate memory is one way to avoid trouble. Overall, in our experience programming remains convenient and flexible when our persistent heap is used in conjunction with STL.

One possible concern with the composition of STL with a persistent heap based on failure-atomic `msync()` is that the latter's `sync()` cost depends on the number of memory pages dirtied between consecutive `sync()` calls, which STL makes no attempt to minimize. This concern is well-founded because containers such as the versatile and powerful STL `<set>` and `<map>` are typically implemented with red-black trees, whose balancing algorithm can make small changes to several memory pages for a single insertion or deletion. We shall revisit STL/persistent-heap performance in Section 5.3.

4.2 Kyoto Tycoon Key-Value Server

Our failure-atomic `msync()` provides a simple, POSIX-compatible programming interface that can be easily retrofitted onto existing software for efficiently preserving the integrity of durable data. We present a case study of adopting failure-atomic `msync()` in the Kyoto Tycoon key-value server [16]. Tycoon encapsulates a lightweight database engine [15] that supports hashing-based key-value insert, replace, delete, and lookup. Tycoon runs an HTTP server that interfaces with (potentially concurrent) web clients. Each record modification operation (insert, replace, or delete) involves changes to both data and the hashing structure. Data integrity requires that each operation must commit to durable storage atomically. Consistency between clients and the server further requires that data from each operation is made persistent before responding to the client.

Tycoon uses a memory mapped region to maintain its data structures. Memory data is committed to the storage under two specific mechanisms. The *Synchronize* mechanism enforces all dirty data in the memory mapped region to be committed to the durable storage through conventional `msync()`. However, the Synchronize mechanism cannot prevent operating system background flushes that may write an inconsistent, partial set of dirty data. Tycoon's *Transaction* mechanism maintains a user-level UNDO log to ensure that each transaction is committed all or nothing despite failure or crashes. During failure recovery, the UNDO log is replayed backwards to roll back partially executed transactions. The Transaction mechanism is expensive due to the additional log writes and the ordering between log and data writes. These mechanisms are selected by setting Tycoon server's startup parameter.

With our failure-atomic `msync()`, data commits in Tycoon's Synchronize mechanism become atomic and therefore data integrity is ensured as long as an atomic data sync is called at the end of each request (right before responding to the client). Failure-atomic `msync()` effectively elevates the cheaper Synchronize mechanism to be as powerful as the more expensive Transaction mechanism in terms of data integrity over failures. It is extremely simple to retrofit failure-atomic `msync()` onto Tycoon. One only has to change a single line of source code—adding our `MAP_ATOMIC` flag at one `mmap()` call in `kcfile.cc`.

5. Experimental Evaluation

We implemented failure-atomic `msync()` in Linux kernel 2.6.33 and evaluated its correctness and performance. We performed experiments on a machine with a quad-core 8-thread 2.40 GHz Intel Xeon E5620 processor and 12 GB memory. Our evaluation uses several mechanical and Flash-based solid-state disks (SSDs) that will be described later. We disable the volatile on-board write caches which cannot survive a power failure. This is necessary for data durability as it prevents the device from prematurely responding to a write request before the written data has been made durable. Our experiments utilize the ext4 file system mounted in `noatime` mode. We use the default Linux CFQ I/O scheduler. Though the CFQ scheduler produces poor fairness on SSDs [33], it is sufficient for our evaluation that focuses on data reliability and performance.

We begin our evaluation of failure-atomic `msync()` by verifying that it does in fact preserve the integrity of user data across the most dangerous type of sudden crashes (Section 5.1). We then measure the performance of our new primitive alone (Section 5.2) and of our composition of failure-atomic `msync()` with the C++ STL (Section 5.3). Finally, we evaluate the performance of our fortified Tycoon server (Section 5.4).

5.1 Data Durability and Consistency over Failures

We evaluated data durability and consistency using experiments with injected power interruptions. Each power interruption test cuts the power to the entire machine, which stresses the reliability of the full system software/hardware stack.

While testing over several storage devices, a surprising by-product of our experiments is that some storage devices do not behave correctly under power failures. Of the five

4 KB Write Latency		
	Sequential	Random
HDD	8.47 msecs	7.03 msecs
SSD1	4.20 msecs	4.61 msecs
SSD2	0.08 msecs	0.11 msecs

Table 1: 4 KB write latency in milliseconds for reliable storage devices, write caches disabled.

SSDs and one hard disk we tested, one SSD permanently failed after our power failure tests. Two other SSDs lose data for writes that completed before the injected power interruptions. These are inherent storage errors unrelated to our failure-atomic `msync()`. We saw similar results from a more comprehensive study indicating that some storage devices do not guarantee data consistency across power failures [47].

We refer to our three reliable storage devices as "HDD," "SSD1," and "SSD2." Using these devices, we evaluated the correctness of failure-atomic `msync()` in ensuring data durability and consistency over failures. We presented a well-defined, synchronous workload using failure-atomic `msync()` to the device and periodically cut power to the entire machine. Upon system boot, the device is checked for data inconsistency before restarting the workload. On each of the three reliable storage devices, we performed several hundred power interruptions without observing any sign of data inconsistency. These tests validate failure-atomic `msync()`'s correctness on reliable storage devices.

5.2 Microbenchmark Evaluation

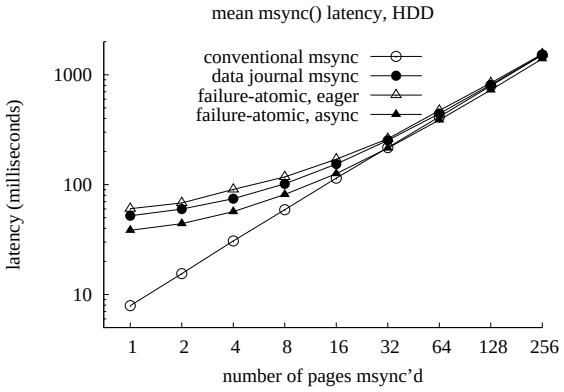
Table 1 presents the latency of synchronous 4 KB writes (queue depth = 1) to the three storage devices. HDD exhibits latency in line with the average seek / rotational latency for a mechanical disk. SSD1 is an earlier-generation Flash-based SSD that exhibits much slower performance than that of a single Flash page program which is on the order of hundreds of microseconds. This is because updating a Flash page causes the SSD firmware, in particular the Flash translation layer, to update the logical-to-physical mapping tables along with any other internal metadata and make those updates durable as well, which incurs several more Flash page writes. SSD2 is a latest-generation Flash-based SSD that, among other optimizations, uses a supercapacitor-backed write cache that survives power failures. Both SSD hardware and proprietary firmware can vary greatly from vendor to vendor, causing a wide variation in observed performance.

We evaluate failure-atomic `msync()` performance against both "conventional `msync()`" and "data journal `msync()`". For the conventional `msync()` we mount the file system journal in the default `ordered` mode. In `ordered` mode, only metadata is journaled. However, the data blocks associated with the inode whose metadata is journaled are written to the device first before committing the metadata. For data journal `msync()`, the file system is mounted in `journal` mode, in which both data and metadata blocks are written to the journal. Note that when ext4 is mounted to use data journaling, file data is journaled but it provides programmers no option of controlling *when* data is journaled. For failure-atomic `msync()`, we also mount the file system in `journal` mode to utilize its failure-atomic properties with our system interface and implementation. We also present results on two versions of failure-atomic `msync()`, one utilizing an asynchronous journaled writeback mechanism and the other performing eager journaled writeback.

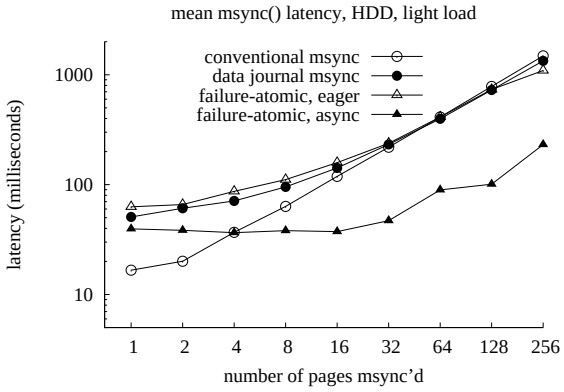
We measure performance by examining latency for `msync()` calls of varying size in two scenarios: a "saturation" scenario in which `msync()` is called repeatedly with no pause between consecutive calls and a "light load" scenario where the test process sleeps for a fixed time following each `msync()` call. For the light load, we use different inter-`msync()` thinktime customized to specific storage devices—longer thinktime for slower I/O device so that the relative stress to the I/O device is balanced. The test memory-maps a file and loads the entire file into memory. A fixed number of random pages are dirtied and then `msync()` is called. The tests were conducted for a minimum of 200 iterations and duration of 90 seconds.

Figure 2 presents *mean* `msync()` latencies for HDD, SSD1, and SSD2 under both saturation and light load as the number of pages per `msync()` increases. Mean latencies reflect the aggregate performance that would be observed over a long sequence of observations. Under saturation load, for small numbers of pages, conventional `msync()` exhibits the lowest latency as it issues fewer I/O requests than `msync()` using data journaling and both variants of failure-atomic `msync()`. As the number of pages per `msync()` increases, similar performance is observed for all systems across all devices as the latency is dominated by large numbers of random writes. Under light I/O load, as the number of pages per `msync()` increases, failure-atomic `msync()` using asynchronous journaled writeback offers the lowest latency of the alternatives. By leveraging asynchronous journaled writeback, failure-atomic `msync()` only performs large sequential I/O, i.e., journal commits, on the critical path; journaled writeback is free to occur during inter-`msync()` thinktime. With `msync()` sizes of as little as four pages, failure-atomic `msync()` is capable of matching or outperforming the standard `msync()`.

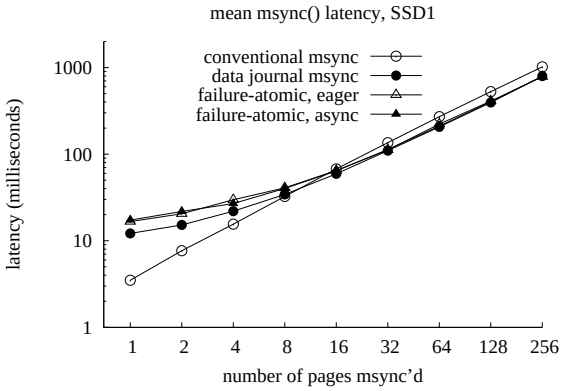
We note that the performance benefit of asynchronous journaled writebacks is less at saturation load compared to its benefit under light load. The reason is simple—delegating some work asynchronously is only beneficial when there is free device time to perform such work at no cost. We also observe that the benefit of asynchronous journaled writebacks is least on the fastest I/O device—SSD2. As explained at the end of Section 3.2, the additional latency of synchronously performing non-critical I/O on fast storage devices may be



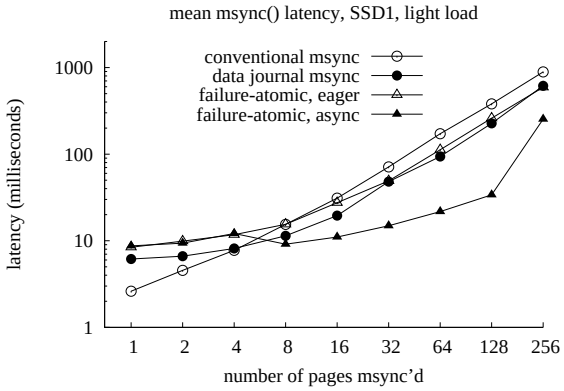
(a) HDD



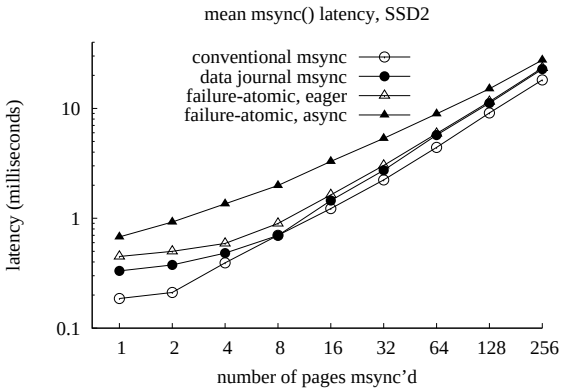
(b) HDD, 4 secs inter-`msync()` thinktime



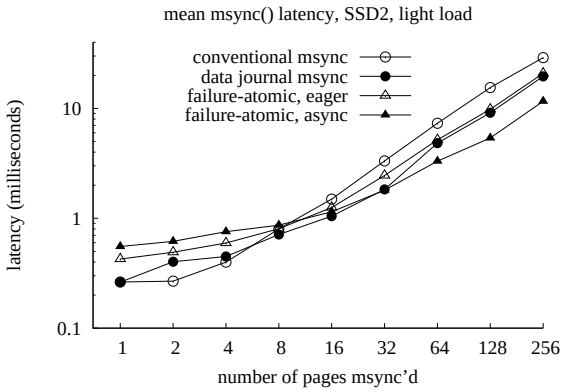
(c) SSD1



(d) SSD1, 1 sec inter-`msync()` thinktime



(e) SSD2



(f) SSD2, 100 msecs inter-`msync()` thinktime

Figure 2: Mean msync() latency as a function of number of dirty pages sync'd.

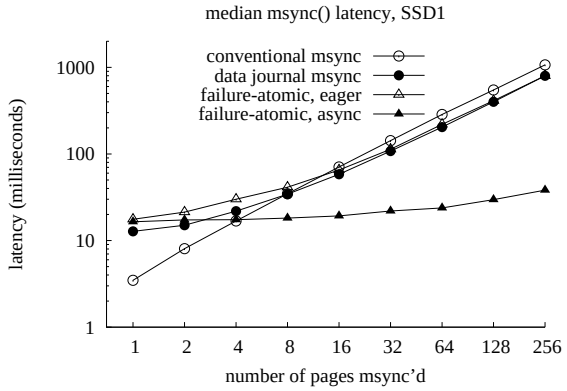


Figure 3: Median latency as a function of number of dirty pages sync'd.

outweighed by the costs of additional page lookups and higher memory pressure.

Figure 3 presents the median latency for saturation workloads running on SSD1. Comparing Figure 3 and Figure 2c, we see that the mean latency is notably higher than the median latency for the asynchronous journaled writeback variant of failure-atomic `msync()`. Recall that the distinction between our two variants is when the random in-place writes are dispatched to the file system. The eager failure-atomic `msync()` will initiate writes on previously journaled pages on the subsequent encounter, i.e., the next `msync()`. While this reduces memory pressure, it suffers from synchronously initiating work that is not semantically critical to the atomic operation and then waiting for unnecessary I/O to complete, resulting in performance similar that of data journaled `msync()`. When failure-atomic `msync()` utilizes asynchronous journaled writeback, it defers writes so that they are not issued on the critical path. While this deferment allows most `msync()` calls to exhibit low latency, a consequence of this deferment is that the pool of dirty but journaled pages can quickly grow large in the background, particularly under saturation load. When journaled writeback is initiated, a significant amount of work is issued to the device, which can interleave and interfere with the sequential journal write for a period of time.

5.3 Persistent Heap and C++ STL

We now return to the simple composition of the C++ Standard Template Library (STL) with a persistent heap based on failure-atomic `msync()`, outlined in Section 4.1. We compare the performance of an STL `<map>` container with a range of alternative local databases capable of storing key-value pairs and capable of supporting transactional updates, with varying levels of feature-richness:

SQLite is a software library that supports conventional relational database manipulation via SQL [40]. Client code links directly with the SQLite library and manipulates data in memory, but may define ACID transactions to storage using an atomic commit mechanism [41]. We use version 3.7.14.1 in our tests.

Kyoto Cabinet is a library that stores key-value pairs in a B+ Tree in memory and allows client code to define ACID transactions to storage [15]. We use version 1.2.76 in our tests.

LevelDB is a key-value storage library by Dean & Ghemawat of Google [12]. LevelDB supports `put(key, value)`, `get(key)`, and `delete(key)` operations, which may be grouped into failure-atomic batches. We use version 1.6.0 in our tests.

Our tests do not include the well-known Berkeley Database (BDB) because our tests run on a journaling file system (ext4), and BDB does not guarantee recovery from sudden crashes on journaling file systems [5, p. 185].

We ran performance comparisons on a conventional spinning disk and two solid-state drives ("HDD", "SSD1", and "SSD2" in Table 1). We configured each library and corresponding client software to perform each data-manipulation operation in failure-atomic synchronous fashion; we otherwise retained all configuration defaults. Each system performed the following sequence of operations on a universe of 1,000 keys: 1) insert all keys along with random 1 KB values; 2) replace the value associated with each key with a different random 1 KB value; and 3) delete all keys. Each of the above three steps visits keys in a different random order, i.e., we randomly permute the universe of keys before each step.

Table 2 presents mean per-operation times in milliseconds for the four libraries considered. We present results using the same set of "thinktimes" between operations as used in the experiments summarized in Figure 2. One prominent difference between the STL `<map>`/failure-atomic `msync()` combination and the other systems is that the former exhibits more consistent mean per-operation latencies than the others. On the HDD, for example, Kyoto Cabinet `insert` is roughly 3× slower than `replace`; LevelDB `delete` is roughly twice as fast as `replace`; and SQLite `insert` is considerably slower than `delete`. Comparable differences are evident on the two solid-state storage devices. Failure-atomic `msync()` performance is more uniform because the STL `<map>` dirties roughly the same number of pages for all three types of operations.

Our system is usually faster than Kyoto Cabinet and SQLite on all three storage devices, often by a factor of 2–4×. Dean & Ghemawat's lean and efficient LevelDB library outperforms all of the other systems by a wide margin, typically beating our system by 3–4× and the others by up to 6–9×. (A comparison between LevelDB and SQLite isn't entirely fair because they represent opposite ends of

	hard disk (HDD)			solid-state (SSD1)			solid-state (SSD2)		
	thinktime 4 secs			thinktime 1 sec			thinktime 100 millisecs		
	insert	replace	delete	insert	replace	delete	insert	replace	delete
STL <map>/persistent heap/ failure-atomic msync()	37.230	35.787	36.039	8.894	8.297	9.376	0.827	0.591	0.784
Kyoto Cabinet	145.057	54.949	83.281	23.188	8.626	13.251	1.659	0.649	0.938
SQLite	111.746	92.887	81.858	18.331	16.629	14.984	1.352	1.307	0.988
LevelDB	22.383	22.306	12.904	3.199	3.332	2.457	0.303	0.293	0.194
	thinktime zero			thinktime zero			thinktime zero		
	insert	replace	delete	insert	replace	delete	insert	replace	delete
STL <map>/persistent heap/ failure-atomic msync()	36.538	37.382	45.017	11.867	10.632	12.455	0.586	0.581	0.690
Kyoto Cabinet	146.763	54.434	92.951	25.541	8.230	12.632	1.488	0.579	0.942
SQLite	117.067	100.089	84.817	22.491	19.024	17.984	1.229	1.128	1.047
LevelDB	19.385	19.669	8.645	4.130	3.837	2.362	0.212	0.220	0.116

Table 2: Mean per-operation timings (milliseconds) with and without thinktimes between consecutive operations.

a functionality spectrum, with the latter providing the full power of transactional SQL.) Our simple composition of STL's <map> with a persistent heap based on failure-atomic msync() holds up rather well, but falls short of LevelDB's performance for the simple reason noted in Section 4.1: The red-black tree beneath the <map> makes no attempt to be frugal with respect to the dirtying of pages during write operations. By contrast, LevelDB implements atomic updates by compact writes to a log file.

5.4 Kyoto Tycoon Key-Value Server

Section 4.2 showed that a single line of source code change allows the Kyoto Tycoon key-value server to adopt our failure-atomic msync(). By using failure-atomic msync(), the Tycoon server can guarantee the integrity of durable data over failures without using its expensive Transaction mechanism. We compare the performance of three data management approaches for Tycoon—its native Synchronize mechanism, its Transaction mechanism, and failure-atomic msync().

Our experiments use the Tycoon server version 0.9.56. We run client(s) from a different machine in the same local-area network. The ICMP ping round-trip latency between the client and server machines is around 190 μsecs. We measure the Tycoon key-value server performance at two conditions. First, we measure the request response time under light load, when a single client issues requests with 10 msecs thinktime between consecutive requests. We also measure the server throughput under high load, when four clients concurrently issue requests without inter-request thinktime (each issues a new request as soon as the previous one is responded).

Figure 4 illustrates the performance of the Tycoon server under different data management approaches. Among the three approaches, only Tycoon Transaction and failure-atomic msync() guarantee data integrity over failures. While Tycoon Transaction performs a similar amount of I/O as failure-atomic msync() (REDO log and corresponding data write), it performs both an UNDO log write and the corresponding data write synchronously. In comparison, failure-atomic msync() does less synchronous work due to our asynchronous journaled writebacks.

Under light load (left column in Figure 4), we find that failure-atomic msync() reduces the response time of Tycoon Transaction by 43%, 33%, and 42% for record insert, replace, and delete respectively on SSD1. Less performance enhancement on record replace is probably due to less work in a replace (compared to a record insert or delete that must also change the key-value hashing structure). On the faster SSD2, the response time reductions are 33%, 26%, 33% for the three operations. Failure-atomic msync() produces less performance advantage on SSD2 because the I/O time is less dominant on a faster storage device and an I/O-accelerating technique would appear less significant.

Under high load (right column of Figure 4), we find that failure-atomic msync() achieves 1.79×, 1.37×, and 1.73× speedup over Tycoon Transaction for record insert, replace, and delete respectively on SSD1. On the faster SSD2, the throughput enhancements are 1.36×, 1.20×, and 1.33× for the three operations.

Furthermore, the performance of failure-atomic msync() is quite close to Tycoon's Synchronize mechanism despite that Tycoon Synchronize does not guarantee data integrity over failures. This can be attributed to the asynchronous

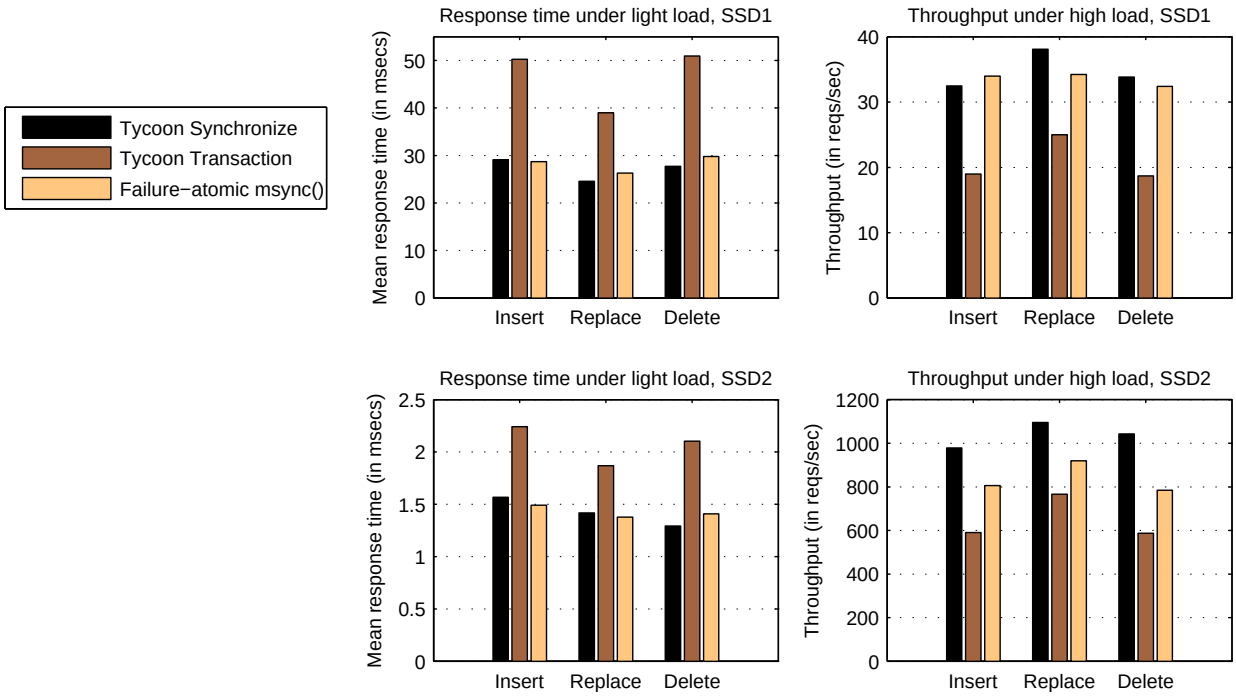


Figure 4: Kyoto Tycoon server insert, replace, and delete performance under different data management approaches on two Flash-based SSDs. The left column shows the response time under light load, when a single client issues requests with 10 msecs thinktime between consecutive requests. The right column shows throughput under high load, when four clients concurrently issue requests without inter-request thinktime (each issues a new request as soon as the previous one is responded).

	Response time under light load			Throughput under high load		
	insert	replace	delete	insert	replace	delete
no-sync	0.47 msecs	0.45 msecs	0.44 msecs	6,646 reqs/s	6,772 reqs/s	7,406 reqs/s
failure-atomic `msync()`	1.49 msecs	1.38 msecs	1.41 msecs	805 reqs/s	919 reqs/s	784 reqs/s

Table 3: Performance comparison (on SSD2) between failure-atomic `msync()`-based Kyoto Tycoon server and the server that makes no explicit effort to commit data to durable storage (thus paying almost no I/O overhead at a large memory configuration).

journaled writeback technique described in Section 3.2—though failure-atomic `msync()` does more I/O work, much of it is not done on the critical application execution path. One exception is that failure-atomic `msync()`'s throughput at high load on SSD2 is clearly slower than Tycoon Synchronize. This is because failure-atomic `msync()`'s asynchronous journaled writeback isn't effective on a highly loaded system with no idle device time for asynchronous work.

Cost of Data Reliability In the final evaluation, we assess the cost of failure-atomic `msync()`-based data reliability compared to a system, called *no-sync*, that makes no explicit effort to commit data to durable storage (thus paying almost no I/O overhead at a large memory configuration). Although the conventional wisdom is that the cost of data reliability is high due to slow I/O, emerging fast I/O devices are gradually reducing this cost. Using SSD2, we compare the performance of the Kyoto Tycoon key-value server under failure-atomic `msync()` and that under no-sync.

Table 3 shows that failure-atomic `msync()` leads to a three-fold response time increase compared to no-sync. Such a cost for data reliability is probably acceptable to many who have feared orders of magnitude performance degradation due to slow I/O. Note that our experiments were performed in a local-area network with a small client-server latency. The cost of data reliability would appear even smaller if the network cost were higher.

However, Table 3 also shows that failure-atomic `msync()` incurs up to nine-fold throughput reduction under high load. This is because no-sync is effectively CPU-bound and its

throughput can benefit from parallel executions over the four CPU cores on our test machine. Similarly, failure-atomic `msync()` would achieve better throughput in a parallel storage array with multiple fast SSDs.

6. Conclusion

We have described the design, implementation, and evaluation of a new mechanism that enables programmers to evolve application data on durable storage from one consistent state to the next even in the presence of sudden crashes caused by power outages or system software panics. Failure-atomic `msync()` is conceptually simple and its interface is a compatible extension of standardized system call semantics. On suitable and appropriately configured storage devices, our implementation of failure-atomic `msync()` in the Linux kernel preserves the integrity of application data from sudden whole-machine power interruptions, the most challenging type of crash failure. Because it writes dirty pages to the journal *sequentially*, the asynchronous-writeback variant of failure-atomic `msync()` is actually *faster* than the conventional `msync()` for updates of medium or large size. Our new mechanism supports flexible and very general higher layers of software abstraction, and it is remarkably easy to retrofit onto complex legacy applications.

Acknowledgments

This work was partially supported by the U.S. Department of Energy under Award Number DE–SC0005026 (see `http://www.hpl.hp.com/DoE-Disclaimer.html` for additional information) and by the National Science Foundation grants CCF-0937571 and CNS-1217372. Kai Shen was also supported by a Google Research Award. We thank Ted Ts'o for several helpful suggestions that guided our implementation strategy, and we thank Joe Tucek for helping us to design our whole-machine power-interruption tests to evaluate the reliability of storage devices. We also thank the anonymous EuroSys reviewers for comments that helped improve this paper. Finally, we thank our shepherd Rodrigo Rodrigues for assistance in the final revision.

References

[1] M. Astrahan, M. Blasgen, K. Chamberlain, K. Eswaran, J. Gravy, P. Griffiths, W. King, I. Traiger, B. Wade, and V. Watson. System R: Relational approach to database management. *ACM Trans. on Database Systems*, 1(2):97–137, June 1976.

[2] M. Atkinson, P. Bailey, K. Chisholm, W. Cockshott, and R. Morrison. PS-Algol: A language for persistent programming. In *Proc. of the 10th Australian National Computer Conference*, pages 70–79, Melbourne, Australia, 1983.

[3] M. Atkinson, K. Chisholm, P. Cockshott, and R. Marshall. Algorithms for a persistent heap. *Software: Practice and Experience*, 13(3):259–271, Mar. 1983.

[4] A. Bensoussan, C. Clingen, and R. Daley. The MULTICS virtual memory: Concepts and design. *Communications of the ACM*, 15(5):308–318, May 1972.

[5] Oracle Berkeley DB Programmer's Reference Guide, 11g Release 2, Dec. 2011. `http://docs.oracle.com/cd/E17076_02/html/programmer_reference/BDB_Prog_%Reference.pdf`.

[6] A. P. Black. Understanding transactions in the operating system context. In *Fourth ACM SIGOPS European Workshop*, pages 73–76, Bologna, Italy, Sept. 1990.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. on Computer Systems*, 26(2), June 2008.

[8] Choi et al. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *Proc. of the Int'l Solid-State Circuits Conf.*, San Francisco, CA, Feb. 2012.

[9] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of the 16th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles (SOSP)*, pages 133–146, Big Sky, MT, Oct. 2009.

[11] De Sandre et al. A 4 Mb LV MOS-selected embedded phase change memory in 90 nm standard CMOS technology. *IEEE Journal of Solid-State Circuits*, 46(1):52–63, Jan. 2011.

[12] J. Dean and S. Ghemawat. leveldb – A fast and lightweight key/value database library by Google. `http://code.google.com/p/leveldb/`.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of the 21st ACM Symp. on Operating Systems Principles (SOSP)*, pages 205–220, Stevenson, WA, Oct. 2007.

[14] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002.

[15] FAL Labs. Kyoto Cabinet: a straightforward implementation of DBM. `http://fallabs.com/kyotocabinet/`.

[16] FAL Labs. Kyoto Tycoon: a handy cache/storage server. `http://fallabs.com/kyototycoon/`.

[17] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. on Computer Systems*, 18(2):127–153, May 2000.

[18] J. F. Garza and W. Kim. Transaction management in an object-oriented database system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 37–45, Chicago, IL, Sept. 1988.

[19] G. V. Ginderachter. V8Ken. https://github.com/supergillis/v8-ken/.

[20] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *Proc. of the USENIX Annual Technical Conf.*, Boston, MA, June 2012.

[21] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2010.

[22] D. Harnie, J. D. Koster, and T. V. Cutsem. SchemeKen. https://github.com/tvcutsem/schemeken.

[23] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. of the USENIX Winter Technical Conf.*, San Francisco, CA, Jan. 1994.

[24] T. Kelly. Ken open-source distribution. http://ai.eecs.umich.edu/~tpkelly/Ken/.

[25] T. Kelly, A. H. Karp, M. Stiegler, T. Close, and H. K. Cho. Output-valid rollback-recovery. Technical report, HP Labs, 2010. http://www.hpl.hp.com/techreports/2010/HPL-2010-155.pdf.

[26] C. Killian and S. Yoo. MaceKen open-source distribution. http://www.macesystems.org/maceken/.

[27] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP)*, pages 92–101, Saint Malo, France, Oct. 1997.

[28] Microsoft. Alternatives to using Transactional NTFS. http://msdn.microsoft.com/en-us/library/hh802690.aspx.

[29] How to have a checkpoint file using mmap which is only synced to disk manually? http://stackoverflow.com/questions/3146496/.

[30] D. Narayanan and O. Hodson. Whole-system persistence with non-volatile memories. In *Proc. of the 17th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, Mar. 2012.

[31] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. *ACM Trans. on Computer Systems*, 26(3), Sept. 2008.

[32] J. O'Toole, S. Nettles, and D. Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proc. of the 14th ACM Symp. on Operating Systems Principles (SOSP)*, pages 161–174, Asheville, NC, Dec. 1993.

[33] S. Park and K. Shen. FIOS: A fair, efficient Flash I/O scheduler. In *Proc. of the 10th USENIX Conf. on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2012.

[34] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles (SOSP)*, pages 161–176, Big Sky, MT, Oct. 2009.

[35] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proc. of the USENIX Annual Technical Conf.*, pages 105–120, Anaheim, CA, Apr. 2005.

[36] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional Flash. In *Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 147–160, San Diego, CA, Dec. 2008.

[37] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In *Proc. of the 14th ACM Symp. on Operating Systems Principles (SOSP)*, pages 146–160, Asheville, NC, Dec. 1993.

[38] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[39] M. I. Seltzer, G. R. Granger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proc. of the USENIX Annual Technical Conf.*, San Deigo, CA, June 2000.

[40] SQLite, Oct. 2012. http://www.sqlite.org/.

[41] SQLite atomic commit, Oct. 2012. http://www.sqlite.org/atomiccommit.html.

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

[43] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of the 9th USENIX Conf. on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2011.

[44] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proc. of the 16th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–103, Newport Beach, CA, Mar. 2011.

[45] S. Yoo, C. Killian, T. Kelly, H. K. Cho, and S. Plite. Composable reliability for asynchronous systems. In *Proceedings of USENIX Annual Technical Conference*, 2012. https://www.usenix.org/system/files/conference/atc12/atc12-final206-7-2%0-12.pdf.

[46] D. H. Yoon, R. S. Schreiber, J. Chang, N. Muralimanohar, P. Ranganathan, and P. Faraboschi. VerMem: Versioned Memory using Multilevel-Cell NVRAM. In *Non-Volatile Memories Workshop*, Mar. 2012. https://lph.ece.utexas.edu/users/dhyoon/pubs/vermem_poster_nvmw12.pdf.

[47] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge. Understanding the robustness of SSDs under power fault. In *Proc. of the 11th USENIX Conf. on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2013.