From ARIES to MARS: Reengineering Transaction Management for Next-Generation, Solid-State Drives

UCSD CSE Technical Report

Joel Coburn Trevor Bunker Rajesh K. Gupta Steven Swanson

Non-Volatile Systems Laboratory
The Department of Computer Science & Engineering
University of California, San Diego
{jdcoburn,tbunker,gupta,swanson}@cs.ucsd.edu

Abstract

Systems that provide powerful transaction mechanisms often rely on write-ahead logging (WAL) implementations that were designed with slow, disk-based systems in mind. The emerging class of fast, byte-addressable, non-volatile memory (NVM) technologies (e.g., phase change memories, spin-torque MRAMs, and the memristor), however, present performance characteristics very different from both disks and flash-based SSDs. This paper addresses the problem of designing a WAL scheme optimized for these fast NVM-based storage systems. We examine the features that a system like ARIES, a WAL algorithm popular for databases, must provide and separate them from the implementation decisions ARIES makes to optimize for disk-based systems. We design a new NVM-optimized WAL scheme (called MARS) in tandem with a novel SSD multi-part atomic write primitive that combine to provide the same features as ARIES does without any of the disk-centric limitations. The new atomic write primitive makes the log's contents visible to the application, allowing for a simpler and faster implementation. MARS provides atomicity, durability, and high performance by leveraging the enormous internal bandwidth and high degree of parallelism that advanced SSDs will provide. We have implemented MARS and the novel visible atomic write primitive in a next-generation SSD. This paper demonstrates the overhead of the primitive is minimal compared to normal writes, and our hardware provides large speedups for transactional updates to hash tables, b-trees, and large graphs. MARS outperforms ARIES by up to 3.7× while reducing software complexity.

1 Introduction

Emerging fast non-volatile memory (NVM) technologies, such as phase change memory, spin-torque transfer memory, and the memristor, are orders of magnitude faster than existing storage technologies (i.e., disks and flash). Such a dramatic improvement shifts the balance between storage, system bus, main memory, and CPU performance and will force designers to rethink storage architectures to maximize application gains and exploit memory performance and parallelism. While recent work focuses on optimizing read and write performance for storage arrays based on these memories [5, 6], systems must also provide strong guarantees about data integrity in the face of failures.

File systems, databases, persistent object stores, and other applications that rely on persistent data structures must provide strong consistency guarantees. Typically, these applications use some form of transaction to move the data from one consistent state to another. Most systems implement transactions using software techniques such as write-ahead logging (WAL) or shadow paging. These techniques are based on complex, disk-based optimizations designed to minimize the cost of synchronous writes and leverage the sequential bandwidth of disk.

NVM technologies provide very different performance characteristics compared to disk, and exploiting them requires new approaches to implementing application-level transactional guarantees. NVM storage arrays provide parallelism within individual chips, between chips attached to a memory controller, and across memory controllers. In addition, the aggregate bandwidth across the memory controllers in an NVM storage array will outstrip the interconnect (e.g., PCIe) that connects it to the host system.

We develop a novel WAL scheme, called MARS, optimized for NVM-based storage. The design of MARS

reflects an examination of ARIES [21], a popular WAL-based recovery algorithm for databases. We separate the features ARIES must provide from the architectural decisions it is built on that optimize for disk-based systems. MARS uses a multi-part atomic write primitive to implement ACID transactions on top of a novel NVM-based SSD architecture. As we will show, multi-part atomic writes simplify the implementation of ARIES by removing disk-centric optimizations. They are also a useful building block for other transaction mechanisms and applications that must provide strong consistency guarantees.

The multi-part atomic write interface supports atomic writes to multiple portions of the storage array without alignment or size restrictions, and the hardware shoulders the burden for logging and copying data to enforce atomicity. This interface exposes the logs to the application and allows the user to manage the log space directly, providing greater flexibility for software to implement transactions. In contrast, recent work on atomic write support for flash-based SSDs [23, 22] hides the logging in the flash translation layer (FTL), restricting user interaction with the logs.

Our design achieves high performance by distributing logging, commit, abort, and recovery functions across multiple memory controllers to leverage the internal bandwidth of the storage device. Shifting support for logging and commit into hardware relieves pressure on the PCIe link and minimizes operating system overhead, since issuing an atomic write requires just a single IO request and a single DMA transfer.

We implemented our design in a prototype PCIe storage array [5]. Microbenchmarks show that our atomic writes reduce latency by 2.9× compared to using normal synchronous writes to implement a traditional logging protocol, and our atomic writes increase effective bandwidth by between 2.0 and 3.8× by eliminating overhead. Compared to non-atomic writes, atomic writes reduce effective bandwidth just 1-8% and increase latency by just 30%.

We use the atomic write primitive to accelerate database-style logging, simple on-disk persistent data structures, and common cloud computing "NoSQL" services. MARS improves performance by 3.7× relative to our baseline version of ARIES. We also implement ACID key-value stores based on a hash table and a B+tree as well as an application that simultaneously updates and queries a large scale-free graph. On average, atomic writes improve performance by 1.4× compared to software-based implementations of the same ACID guarantees and reduce performance by only 15% compared to non-transactional versions. Finally, we integrate our key-value store implementation into MemcacheDB [8], replacing Berkeley DB, and find that using atomic writes increases performance by up to 3.8×.

The remainder of this paper is organized as follows. In Section 2, we describe the memory technologies and storage system that our work targets. Section 3 examines existing transaction mechanisms in the context of fast NVM-based storage, deconstructs ARIES, and proposes MARS as an alternative for fast NVM-based storage. In Section 4, we describe a new set of IO primitives that support multi-part atomic write operations for MARS and other applications. Section 5 places this work in the context of prior work on support for transactional storage. Section 6 describes the hardware architecture in detail. Section 7 evaluates our multi-part atomic write support and its impact on the performance of MARS and other persistent data structures. Section 8 summarizes our contributions.

2 Storage technology

Fast NVMs will catalyze changes in the organization of storage arrays and how applications and the OS access and manage storage. Storage will soon look and behave more like DRAM than flash memory, and this motivates our redesign of transaction mechanisms to take advantage of fast NVMs with a novel multi-part atomic write interface. This section describes the memories and architecture of the storage system that our work targets. Section 6 describes our implementation in more detail.

Fast non-volatile memories such as phase change memories (PCM) [4], spin-torque transfer [13] memories, and memristor-based memories differ fundamentally from conventional disks and from the flash-based SSDs that are beginning to replace them. NVMs' most important features are their performance (relative to disk and flash) and their simpler interface (relative to flash).

Predictions by industry [16] and academia [4, 13] suggest that NVMs will have bandwidth and latency characteristics similar to DRAM. This means they will be between 500 and 1500× faster than flash and 50,000× faster than disk.

Our baseline storage array is the Moneta SSD [5]. It spreads 64 GB of storage across eight memory controllers connected via a high-bandwidth ring. Each memory controller provides 4 GB/s of bandwidth for a total internal bandwidth of 32 GB/s. An 8-lane PCIe 1.1 interface provides a 2 GB/s full-duplex connection (4 GB/s total) to the host system. The prototype runs at 250 MHz on a BEE3 FPGA prototyping system [2].

The Moneta storage array emulates advanced non-volatile memories using DRAM and modified memory controllers that insert delays to model longer read and write latencies. We model phase change memory (PCM) in this work and use the latencies from [18] (48 ns and 150 ns for array reads and writes, respectively).

Unlike flash, PCM (as well as other NVMs) does not

require a separate erase operation to clear data before a write. This makes in-place updates possible and, therefore, eliminates the complicated flash translation layer that manages a map between logical storage addresses and physical flash storage locations to provide the illusion of in-place updates. PCM still requires wear-leveling and error correction, but fast hardware solutions exist for both of these in NVMs [24, 25, 27]. Moneta uses start-gap wear leveling [24]. With fast, in-place updates, Moneta is able to provide low-latency, high-bandwidth access to storage that is limited only by the interconnect (e.g. PCIe) between the host and the device.

3 Revisiting Transaction Support

This section examines existing transaction mechanisms, focusing on ARIES write-ahead logging. We describe the features that a system like ARIES must provide to implement flexible ACID transactions. We analyze the design decisions that make ARIES a good fit for disk but are not well-suited for fast NVM-based storage. Then, we propose MARS, a novel WAL architecture that takes advantage of the multi-part atomic writes provided by our prototype storage array. MARS provides the features required by ARIES while exploiting the performance of fast NVMs.

3.1 Transaction mechanisms

Transaction implementations differ depending on the requirements of the application and the underlying storage technology. For many applications, relational databases are a good fit because they provide full ACID semantics and accomodate a wide variety of data formats and operations on that data. Many databases are built on top of ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) [21], a powerful algorithm for providing strong consistency guarantees. ARIES-style transactions are scalable and support different levels of isolation.

For web services or file systems, simpler approaches are often the best option because the class of transactions they must support is narrower. Transaction size is often fixed or bounded, and transactions often need not have the flexibility to read back or update data multiple times. Previous systems [14, 23, 22] provided an atomic write interface that is limited to these types of transactions. The systems batched writes up in memory and sent them to the storage array in a single IO operation. Our multi-part atomic write interface, on the other hand, allows transactions to be specified in multiple IO requests, offering scalability and better programmability. Also, this interface provides visibility to each logged part of a transaction prior to commit.

Transaction implementations typically include a concurrency control scheme (e.g. two-phase locking [3]), some form of data versioning, and a recovery algorithm. The most common ways to implement data versioning are either by using write-ahead logging and updating data in-place or by using shadow paging, techniques that optimized heavily for disk. With this in mind, we now re-examine existing transaction mechanisms in the context of fast NVM-based storage and the high-level features that applications demand.

3.2 Deconstructing ARIES

We focus on ARIES because it influenced the design of many industrial-strength databases and is a key building block in providing fast, flexible, and efficient ACID transactions. ARIES uses WAL and has been tuned to exploit the sequential write performance of disk. In Table 1, we list several of the important *features* that ARIES provides to higher-level software (e.g., the rest of the database) and that make it useful to a variety of applications. For example, ARIES offers flexible storage management since it supports objects of varying length. It also allows transactions to scale with the amount of free disk storage space rather than with available main memory. Features like operation logging and fine-grained locking improve concurrency. Recovery independence makes it possible to recover some portion of the database even when there are errors. Independent of the underlying storage technology, ARIES must export these features to the rest of the database.

To provide these features and achieve high performance, ARIES incorporates a set of *design decisions* (Table 2) that exploit the properties of disk: They optimize for long, sequential accesses and avoid short, random accesses whenever possible. Also, because disk drives are effectively serial devices, these design decisions optimize for a single write stream. These design decisions are a poor fit for advanced, solid-state storage arrays which provide fast random access, high internal bandwidth, and a high degree of parallelism. Below, we describe the design decisions ARIES makes that optimize for the properties of disk and show how they limit the performance of ARIES on an NVM-based storage device.

No-force In ARIES, the system writes log entries to the log in storage before any changes to an object are written to storage. Then, if a crash occurs, ARIES can redo the partially completed operation. To hide the latency of random writes to disk, ARIES implements a *no-force* policy, which means the system writes updated pages back to disk after commit. ARIES flushes redo log entries to disk in a synchronous sequential write during commit, making the updates available for the recovery routine to reapply in case of a failure. In fast NVM-based storage, how-

Feature	Benefits	Available in MARS?
Flexible storage management	Supports varying length data	Yes
Fine-grained locking	High concurrency	Yes
Partial rollbacks via savepoints	Robust and efficient transactions	Yes
Operation logging	High concurrency lock modes	Out of scope
Recovery independence	Simple and robust recovery	Out of scope

Table 1: **ARIES features** ARIES-style WAL provides the above features to the rest of the system regardless of storage technology.

Design option	Advantage for disk	Implementation	Alternative for MARS
No-force	Eliminate synchronous random writes	Flush redo log entries to storage on commit	Force in hardware at memory controllers
Steal	Reclaim buffer space Eliminate random writes Avoid false conflicts	Write undo log entries before writing back dirty pages	Hardware does in-place updates Log always holds latest copy
Pages	Simplify recovery and buffer management	Perform updates on pages Page writes are atomic	Hardware uses pages Software operates on objects
Log Sequence Numbers (LSNs)	Simplify recovery Enable high-level features	Order updates to storage using LSNs	Hardware enforces ordering with commit sequence numbers

Table 2: **ARIES design decisions** ARIES relies on a set disk-centric optimizations to maximize performance on conventional storage systems. However, these optimizations are a poor fit for the characteristics of storage based on fast, non-volatile memories.

ever, random writes are no more expensive than sequential writes, so the value of a no-force policy is much lower.

Steal A *steal* policy allows the buffer manager to write dirty pages back to disk before commit. While stealing greatly improves the performance of ARIES running on disk, it provides little to no benefit for fast NVM-based storage. By writing pages back early, the buffer manager can reclaim buffer space during transaction execution (supporting larger transactions), group writes together to take advantage of sequential disk bandwidth, and avoid data races on pages shared by overlapping transactions. Stealing requires undo logging because it is only safe to write back dirty pages if copies of old values have been written to disk. After a crash or abort, the system may use the undo log entries to recreate the overwritten data.

For disk, the performance benefits greatly outweigh the overhead of the extra logging. With fast NVMs, because the performance of random writes and sequential writes is the same, the overhead of undo logging can actually hurt overall performance. The cost of writing a log entry to storage before making an update occurs on every udpate, but the benefit of writing pages back early occurs far less frequently. While stealing eliminates costly seek time for disk, writing pages back early as part of a larger write to fast NVM-based storage only helps amortize the setup/completion cost of an IO request.

Pages and LSNs In ARIES, disk pages are the basic unit of recovery and each page contains a log sequence number (LSN). LSNs provide an ordering on disk updates. At recovery, ARIES uses LSNs to decide which updates to reapply to bring the system into a consistent state. While the design of ARIES is not restricted to pages per se, pages simplify the implementation of recovery. The system assumes single page writes are atomic and uses them as a foundation for larger atomic writes. When the system logs an update, it writes the LSN in the same page as its matching log record, guaranteeing that the two are updated atomically. To be useful for recovery, LSNs must be generated with a unique order and must be written out to disk in that order [17]. This adversely affects performance. It also complicates situations where objects span multiple pages or multiple objects fit in a single page. Recent work [29] proposes segments as an alternative to pages, making it possible to efficiently handle objects of various sizes and copy them directly between the application and storage array.

Pages and LSNs are even more restrictive for fast NVM-based storage arrays because they limit parallelism, waste bandwidth, and increase latency. Maintaining headers in log records and forcing log records out to disk in LSN order serializes execution, resulting in underutilization of the storage array. Because objects may share pages, LSNs may artificially order updates when the system could in fact perform those updates in parallel. Also, when objects consume less than a page of storage, the sys-

tem must pay the additional cost in IO processing to update an entire page. This is particularly wasteful because fast NVM-based storage has no sector or page restriction on access size and can handle arbitrarily-sized requests efficiently.

3.3 Building MARS

We now describe the design of MARS, an alternative transaction mechanism based on ARIES but adapted to the characteristics of fast NVM-based storage. MARS relies on our multi-part atomic write primitive, presented in Section 4, and ensures that the most recent copy of an object is always directly accessible, whether the most recent copy lives at the object's home location or somewhere in a log. Using multi-part atomic writes, MARS can eliminate the need for pages and LSNs. MARS replaces the no-force and steal policies designed for disk with more efficient mechanisms that utilize the internal bandwidth of the storage array and the flexible interface of our IO primitive. For each design option in Table 2, we propose an alternative method better suited to fast NVM-based storage.

No-force Instead of performing in-place update asynchronously from software, we implement a *force* policy in hardware at the memory controllers. This takes advantage of Moneta's large internal bandwidth—32 GB/s at the memory controllers compared to 4 GB/s PCIe link bandwidth—and eliminates the extra IO requests required for commits and write backs. Also, moving write backs into hardware has other benefits: It eliminates the need for checkpointing the log, and the system immediately reclaims the log space. We choose a force policy over no-force because it allows our hardware to utilize idle cycles, make better use of limited hardware transaction resources, and minimize the amount of work needed to be done at recovery time.

Steal Instead of writing dirty pages back early, we propose simply dropping pages from the buffer pool to acquire free space when needed. Our multi-part atomic write architecture makes this possible: The system first writes an update out to the log and then proceeds to update the buffer pool page. Unlike other systems [14, 23, 22], we do not wait to flush the log at commit. Consequently, the system can page in the updates from the log later as needed. To do this, the system must maintain a mapping of buffer pool pages to log entries, which is possible using our atomic write interface because software controls the placement of log entries in the log files.

Pages and LSNs Because fast NVM-based storage directly supports updates of arbitrary sizes and our IO primitive makes those updates atomic, MARS can eliminate the use of pages as the basic unit of update. Instead, MARS can maintain the same contiguous layout of application-level objects in both storage and main memory. This has two advantages. First, it avoids the cost of translating objects back and forth between pages and their native, in-memory format. Second, because software no longer needs to intervene on a per-page basis, it enables the use of DMA and zero-copy IO operations [29].

Our multi-part atomic write interface eliminates the need for software managed and enforced LSNs. Instead, the storage array maintains ordering in hardware by assigning a unique commit sequence number to a transaction at commit time. This effectively removes the serialization of write requests due to LSNs, allowing log writes from different transactions to proceed in parallel.

With an implementation based on multi-part atomic writes, MARS provides the features (shown in Table 1) that ARIES exports to higher-level software while significantly reducing software complexity. MARS provides flexible storage management and fine-grained locking by making objects directly accessible. Partial rollbacks are achieved using an abort function provided by our hardware that can rewind to any point in the log. Operational logging and recovery independence are currently out of the scope of our atomic write primitive, requiring customizations to the interface specific to ARIES. However, they are a possible topic for future work.

4 Multi-part atomic write architecture

To take advantage of storage array architectures based on fast NVMs, we present a novel multi-part atomic write interface that provides efficient and safe updates to storage. Multi-part atomic writes offer a simple, flexible, and general-purpose way to implement transactions at the application-level. This section describes our atomic write interface, highlighting how transactions execute and how the interface makes the log visible to the application. We discuss the rationale behind our design.

4.1 The transaction model and interface

Our system provides the means to group multiple write operations into transactions and ensure they execute atomically and durably. To achieve full ACID semantics, the application implements consistency and isolation in software. The writes in a transaction can be scattered throughout the storage array and be of any size or alignment. The total size of data that a transaction can update is limited only by the space available for storing the log in the storage array.

Applications create and execute transactions using the commands in Table 3. Each application accessing the

Command	Description
LogWrite(TID, file, offset, data, len, logfile, logoffset)	Record a write to the log at the specified log offset. After commit, copy the data to the offset in the file.
Commit(TID)	Commit a transaction.
Abort(TID) Abort(TID, logfile, logoffset)	Cancel the transaction entirely, or perform a partial rollback to a specified point in the log.
AtomicWrite(TID, file, offset, data, len, logfile, logoffset)	Create and commit a transaction containing a single write.

Table 3: **Multi-part atomic write commands** These commands allow the application to perform atomic and durable updates to the storage array. `LogWrite` returns a TID to the user that must be used on subsequent operations in the same atomic write.

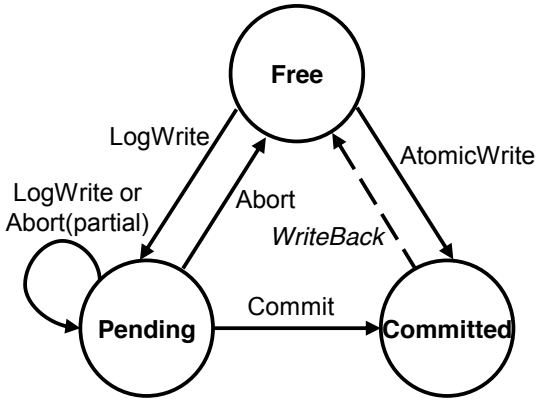


Figure 1: **Transaction state diagram** The system tracks the state of each transaction to guarantee that updates are atomic and durable.

storage device has a private set of 64 transaction IDs (TIDs), and the application is responsible for tracking which TIDs are in use. The commands in the table move a transaction between three possible states: FREE, PENDING, or COMMITTED (Figure 1).

To create a new transaction with TID T, the application issues a `LogWrite` command with T as the first parameter. `LogWrite` records the data, size, and target location for the write in a log. The user provides a log file descriptor and a log offset to indicate the desired position to store the log record. This operation does not actually modify the contents of the target location. After the first `LogWrite`, the state of the transaction changes from FREE to PENDING, indicating that the transaction is in progress but not committed. Additional calls to `LogWrite` add new writes to the transaction.

The writes in a transaction are not visible to other transactions until after commit. However, the transaction can see its own writes prior to commit by keeping track of the log offsets that it associates with each piece of data. After an initial log write for a particular piece of data, a transaction may update that data again before commit simply by writing to the correct log location.

To complete a transaction, the application issues `Commit`(T). The storage array assigns the transaction a commit sequence number that determines the commit order of this transaction relative to others. When the command completes, the transaction has logically committed, and the transaction moves to the COMMITTED state. If a failure should occur after a transaction logically commits but before the system writes the data back, then the recovery mechanism will replay the log to successfully complete the in-place updates.

The hardware can notify the application that the `Commit` is complete before the hardware copies the contents of log into their target locations, but during the commit process, reads and writes to the affected areas stall. This ensures that from the perspective of any application accessing the storage array, commit occurs atomically. When the copy is complete, the TID returns to FREE and the hardware notifies the application that the transaction finished successfully. At this point, it is safe to read the updated data from its target locations.

The application can also `Abort` a transaction, freeing any log entries associated with it and returning it to FREE. Our model supports partial rollbacks of transactions by allowing the user to specify an `Abort` command with an offset into the log. The log offset acts as a savepoint: Any log entries starting from the log offset and going up through the most recent log entry log will be freed, effectively canceling those updates.

Our system provides flexibility by allowing the application to specify atomic write operations in multiple parts. However, this interface adds some overhead because each operation requires a separate IO request. To mitigate this cost, `AtomicWrite` combines `LogWrite` and `Commit` requests into a single request, allowing the system to quickly execute transactions that comprise a single write or to avoid the separate `Commit` when it can identify the final write in a transaction.

The system stores the logs as regular files in the storage array, and the logs may expand or shrink in size as the working sets of transactions demand. Conventional stor-

age systems must allocate space for logs as well, but they often use separate disks to improve performance. Our system relies on the log being internal to the storage device, since our performance gains stem from utilizing the internal bandwidth of the storage array's independent memory banks.

The application manages log space by operating on the log files directly with POSIX file IO. For example, the log file can be extended by writing past the end of the file with `write()`, and the log can be truncated with `ftruncate()`.

4.2 Design rationale

Our simple multi-part atomic write model strikes a balance between implementation complexity and functionality. Our model does not provide full ACID transactions, only atomicity and durability. In particular, our system does not provide isolation between transactions or any locking facilities to mediate access to shared data. The application must implement those if needed. However, our system does provide facilities (e.g., updateable log entries) to make implementing these features easier by letting the application access and manage the log space directly. Consequently, transactions may grow in size as needed and they see the results of their own previous but uncommitted updates. This is a key feature for supporting scalable ARIES-style transactions in MARS.

The algorithm our implementation uses to manage and commit transactions is simple. We use redo logging alone and always update the target location on commit (i.e., we use no-steal and force policies in the memory controllers). The high internal bandwidth of our storage array and NVMs' fast random access performance minimizes the impact of using such a simple logging protocol. It also simplifies the hardware, since replaying the logs of committed transactions is sufficient for recovery. Finally, it avoids the remapping of addresses in hardware that a steal or no-force policy would require to hide uncommitted updates.

We could implement a more complex transaction model with conflict detection, locking, roll back, etc., but crafting a one-size-fits-all solution to those problems is not possible. Atomic writes, however, can accelerate and/or simplify ARIES- and shadow-page-style schemes that provide full-fledged ACID transactions in existing applications. Section 7.2 explores and evaluates this idea in detail.

5 Related Work

Atomicity and durability are critical to storage system design, and system designers have explored many different approaches to providing these guarantees. These include approaches targeting disks, flash-based SSDs, and non-volatile main memories (i.e., NVMs attached directly to the processor) using software, specialized hardware, or a combination of the two. Below, we describe existing systems in this area and highlight the differences between them and the system we describe in this work.

5.1 Disk-based systems

Most disk-oriented systems provide atomicity and durability via software with minimal hardware support. Many systems use ARIES-style [21] write-ahead logging to provide durability, atomicity, and to exploit the sequential performance that disks offer. Our system uses write-ahead logging at the memory controllers. ARIES-style logging is ubiquitous in storage and database systems today.

Recent work on segment-based recovery [29] revisits the design of write-ahead logging for ARIES with the goal of providing efficient support for application-level objects. By removing LSNs on pages, segment-based recovery enables DMA or zero-copy IO for large objects and request reordering for small objects. Our system can take advantage of the same optimizations because the hardware manages logs without using LSNs and without modifying the format or layout of logged objects.

Traditional implementations of write-ahead logging are a performance bottleneck in databases running on parallel hardware. Aether [17] implements a series of optimizations to lower the overheads arising from frequent log flushes, log-induced lock contention, extensive context switching, and contention for centralized, in-memory log buffers. Fast NVM-based storage only exacerbates these bottlenecks, but our system eliminates them almost entirely. Because we offload logging to hardware, we remove lock contention and the in-memory log buffers. With fast storage and a customized driver, our system minimizes context switching and log flush delays.

Stasis [28] uses write-ahead logging to support building persistent data structures. Stasis provides full ACID semantics and concurrency for building high-performance data structures such as hash tables and B-trees. It would be possible to port Stasis to use our atomic write support, but achieving good performance would require significant changes to its internal organization.

Our system provides atomicity and durability at the device level. The Logical Disk [12] provides a similar interface and presents a logical block interface based on atomic recovery units (ARUs) [14] – an abstraction for failure atomicity for multiple writes. Like our system, ARUs do not provide concurrency control. Unlike our system, ARUs do not provide durability, but they do provide isolation.

File systems including WAFL [15] and ZFS [11] use

shadow paging to perform atomic updates. Although fast NVMs do not have the restrictions of disk, the atomic write support in our system would help make these techniques more efficient. Recent work on BPFS [10] extends shadow paging to work in systems that support finer-grain atomic writes. They target non-volatile main memory (see below), but our atomic write support could implement their scheme as well.

Researchers have provided hardware-supported atomicity for disks. Mime [7] is a high-performance storage architecture that uses shadow copies for this purpose. Mime offers sync and barrier operations to support ACID semantics in higher-level software. Like our system, Mime is implemented in the storage controller, but its implementation is more complex since it maintains a block map for copy-on-write, and maintains more metadata to keep track of the resulting versions.

5.2 Flash-based SSDs

Flash-based SSDs offer improved performance relative to disk, making latency overheads of software-based systems more noticeable. They also include complex controllers and firmware that use remapping tables to provide wear-leveling and to manage flash's idiosyncrasies. The controller provides a natural opportunity to provide atomicity and durability guarantees, and several groups have done so.

Transactional Flash (TxFlash) [23] extends a flash-based SSD to implement atomic writes in the SSD controller. TxFlash leverages flash's fast random write performance and the copy-on-write architecture of the FTL to perform atomic updates to multiple, whole pages with minimal overhead using "cyclic commit." In contrast, fast NVMs are byte-addressable and SSDs based on these technologies can efficiently support in-place updates. Consequently, our system logs and commits requests differently and the hardware can handle arbitrarily sized and aligned requests.

Recent work from FusionIO [22] proposes an atomic-write interface in a commercial flash-based SSD. Their system uses a log-based mapping layer in the drive's FTL, but it requires that all the writes in one transaction be contiguous in the log. This prevents them from supporting multiple, simultaneous transactions.

5.3 Non-volatile main memory

The fast NVMs that our system targets are also candidates for non-volatile replacements for DRAM, potentially increasing storage performance dramatically. Using non-volatile main memory as storage will require atomicity guarantees as well, and several groups explored options in this space.

Recoverable Virtual Memory (RVM) [26] provides persistence and atomicity for regions of virtual memory. It buffers transaction pages in memory and flushes them to disk on commit. RVM only requires redo logging because uncommitted changes are never written early to disk, but RVM also implements an in-memory undo log so that it can quickly revert the contents of buffered pages without rereading them from disk when a transaction aborts. Rio Vista [19] builds on RVM but uses battery-backed DRAM to make stores to memory persistent, eliminating the redo log entirely. Both RVM and Rio Vista are limited to transactions that can fit in main memory.

More recently, Mnemosyne [30] and NV-heaps [9] provide transactional support for building persistent data structures in byte-addressable, non-volatile memories. Both systems map NVMs attached to the memory bus into the application's address space, making it accessible by normal load and store instructions. Our atomic write hardware support could help implement a Mnemosyne- or NV-Heaps-like interface on a PCIe-attached storage device, but the details of the implementation would be very different.

6 Implementation

In this section, we present the details of the implementation of our interface described in Section 4, including how we issue commands to the array, how our software layer makes logging flexible and efficient, and how the hardware implements a distributed scheme for redo logging, commit, and recovery. We also discuss testing the system.

6.1 Software support

To make logging transparent and flexible, we leverage the existing software stack. First, we extend a user-space driver to implement our transaction API (see Section 4). In addition, we utilize the file system to manage the logs, exposing them to the user and providing an interface that lets the user dictate the layout of the log in storage.

User-space driver Our SSD provides a highly-optimized (and unconventional) interface for accessing data [6]. It provides a user-space driver that allows the application to communicate directly with the array via a private set of control registers, a private DMA buffer, and a private set of 64 tags that identify in-flight operations. To enforce file protection, the user space driver works with the kernel and the file system to download extent and permission data into Moneta, which then checks that each access is legal. As a result, accesses to file data do not involve the kernel at all in the common case. Modifications to file metadata still go through the kernel. Applications can use the new interface without modification,

or even recompilation, since the library interposes on file access calls via LD_PRELOAD. The user space interface lets Moneta perform IO operations very quickly: 4 kB reads and writes execute in $\sim$7 μs.

Our system uses this user space interface and includes the hardware permission checks on file accesses. We modify the user-space driver to implement our transaction API and provide each application with a private set of 64 virtual transaction IDs (TIDs), eliminating synchronization across applications. As a result, applications issue `LogWrite`, `Commit`, `Abort`, and `AtomicWrite` requests to the storage array from user space, avoiding costly interaction with the operating system.

File system managed logs Our system creates and manages a log for each transaction through the file system and exposes these logs to the user. Our system uses two types of files to maintain a log: a *log file* and a *metadata file*.

The log file contains redo data as part of a transaction from the application. The user creates a log file and can extend or truncate the file, based on the application's log space requirements, using regular file IO.

The metadata file records information about each update including the target location for the redo data upon transaction commit. A trusted process called the *metadata handler* creates and manages metadata files. By communicating with this process, the user space driver can "install" and "remove" metadata files for the channel as the sizes of transactions scale. An install operation allocates space for the system to record transaction metadata. When an application ends, the user space driver removes all installed metadata files, releasing them back to the metadata handler.

The system protects the metadata files from modification by an application. If a user could manipulate the metadata, the log space could become corrupted and unrecoverable. Even worse, the user might direct the hardware to update arbitrary storage locations, circumventing the protection of the OS and file system.

Separating the metadata from the redo data allows applications to access the redo data in the same manner as accessing regular application data. In addition, using files to store metadata enables scalable transactions by obviating the need for partitioning the storage device.

To take advantage of the parallelism and internal bandwidth of Moneta, the user space driver ensures the data offset and log offset for `LogWrite` and `AtomicWrite` requests, when translated from logical to physical addresses, target the same memory controller in the storage array. We can make this guarantee by forcing the file system to allocate space in extents that are aligned to and in multiples of Moneta's 64 kB stripe width. With XFS, we achieve this by setting the stripe width parameter (meant for RAID devices) with `mkfs.xfs`.

6.2 Hardware support

The implementation of our atomic write interface divides functionality between two types of hardware components. The first is a logging module, which we call the *logger* (the gray boxes to the right of the dashed line in Figure 2), that resides at each of the system's eight memory controllers and handles logging for the local controller. The second is a set of modifications to the central controller (the gray boxes to the left of the dashed line in Figure 2) that orchestrates operations across the eight logging modules. Below, we describe the layout of the log and the components and protocols the system uses to coordinate logging, commit, and recovery.

Log structure Figure 3 shows an example log for a transaction at a logger. An entry in a transaction table points to an entry in a metadata file. Each metadata entry contains information about an entry in a log file and a pointer to the next metadata entry.

When the metadata handler installs a metadata file, the hardware divides it into 32 B *metadata entries*. Each metadata entry contains information about a log entry. Each metadata entry contains an address to the next metadata entry for the same transaction. Therefore, the log for a particular transaction is simply a linked list of metadata entries.

The system reserves a small portion (2 kB) of the storage at each memory controller for a transaction table. The transaction table stores the state for up to 64 transactions. Each entry in the transaction table includes the status of the transaction, a sequence number, the address of the head metadata entry in the log, and the number of entries in the log.

Distributed logging Each logger module independently performs logging, commit, and recovery operations and handles accesses to the 8 GB of NVM storage at the memory controller.

The logger implements `LogWrite`, `AtomicWrite`, `Commit`, `Abort`, and log recovery operations. The logger sits between the ring interface and the memory controller, allowing it to intercept operations and issue requests to the memory controller to manipulate log data and metadata.

Before an application can make a `LogWrite` or `AtomicWrite` request, it must first direct the metadata handler to install a metadata file. The logger maintains a free list of metadata entries for each channel in storage. When the logger divides the metadata file into metadata entries, it creates a linked list of metadata entries in storage by writing the pointer field for each metadata entry. The logger only maintains the points to the head and tail of the free list for each channel and, therefore, can scale with the number of metadata entries.

To begin a new transaction, an application must

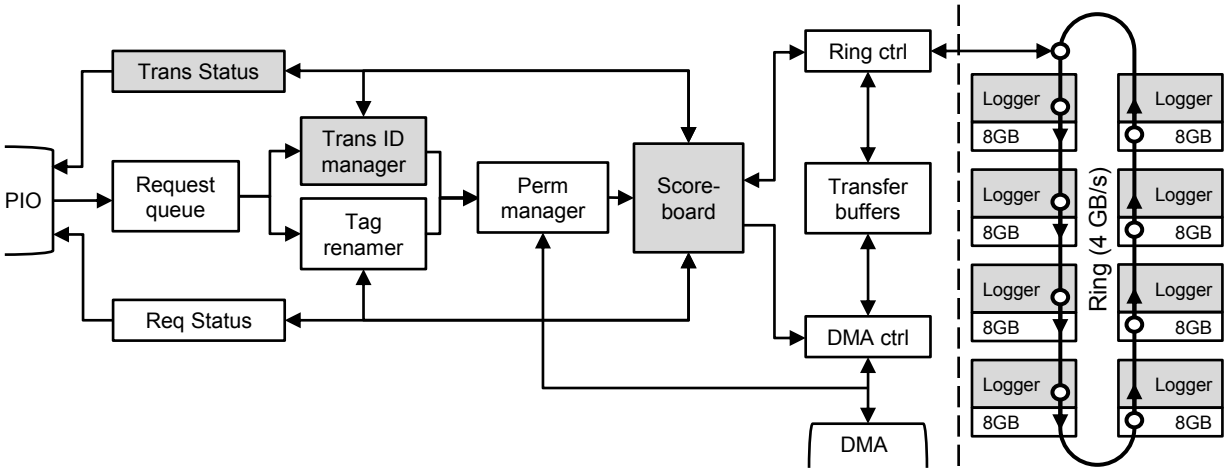


Figure 2: **SSD controller architecture** To support transactions, our system adds hardware support (gray boxes) to an existing prototype SSD. The main controller (to the left of the dotted line) manages transaction IDs and uses a scoreboard to track the status of in-flight transactions. Eight loggers perform distributed logging, commit, and recovery at each memory controller.

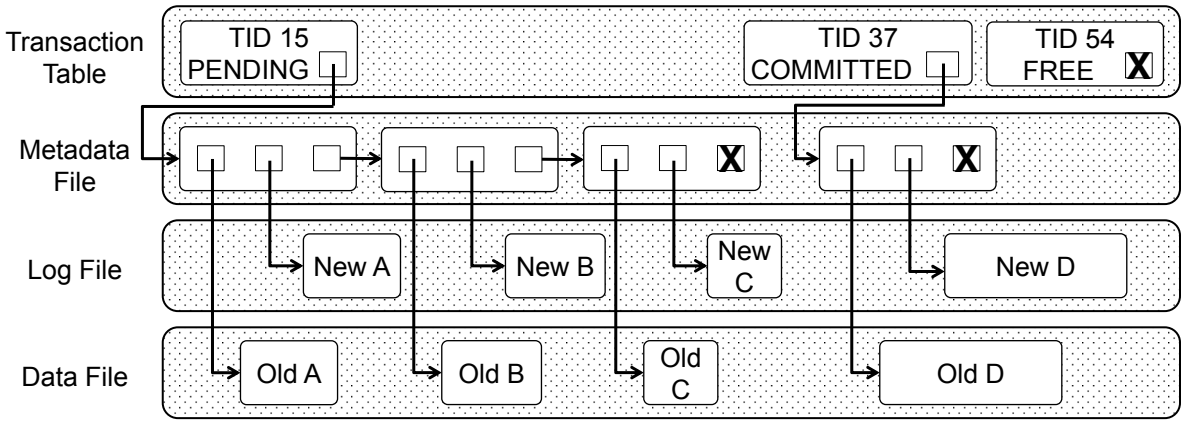


Figure 3: **Example log layout at a logger** Each logger tracks the status of outstanding transactions using the transaction table. Each *TID* has a corresponding transaction table entry. The transaction table entry holds the current state of the transaction, as well as a pointer to the first metadata entry in the log. A metadata entry contains three pointers: a pointer to the address in the data file where the logger will write the redo data, a pointer to the redo data in the log file, and a pointer to the next metadata entry. The logger logically constructs a per-transaction log by linking metadata entries. In this example, *TID 15* has three entries in its log and is still in the PENDING state. *TID 37* is in the COMMITTED state with only a single entry and it is waiting for the logger to write the redo data back to the data file. Finally, *TID 54* is in the FREE state.

have access to a previously created log file. As mentioned in Section 6.1, the application maintains the log file and specifies a log offset for each `LogWrite` or `AtomicWrite` request.

For each `LogWrite` request, the logger allocates a metadata entry, copies the data to the log offset, records the request information in the metadata entry, and then appends the metadata entry to the log.

For an `AtomicWrite` operation, the logger writes the data to the log and immediately marks the transaction COMMITTED, avoiding the extra delay required to coordinate across multiple memory controllers (see next subsection).

The logger implements `Commit` by waiting for all outstanding writes to the log area to complete and then marking the transaction as COMMITTED.

There are two kinds of `Abort` operations that the logger performs: *complete* and *partial*. For a complete `Abort` operation, the logger clears the transaction status and deallocates the metadata entries. On a partial `Abort`, the logger frees the metadata entries until the specified savepoint and makes the savepoint the new head of the transaction's metadata linked list.

A transaction is fully committed when all loggers have marked the transaction as COMMITTED in their transaction tables. The central controller (see next subsection) then directs each logger to apply their respective log. To apply the log, the logger reads each metadata entry in the log linked list. The transaction table indicates the metadata entry at the head of the linked list, as well as the number of entries in the list. For each metadata entry, the logger copies the redo data from the log offset to its destination address. During log application, the logger suspends other read and write operations to make log application appear atomic. At the end of log application, the logger deallocates the transaction's metadata entries.

Since logging and data updates occur locally at each memory controller, logging and commit bandwidth scale with the number of controllers.

The central controller A single transaction may require the coordinate efforts of one or more memory controllers. The central controller (the left hand portion of Figure 2) coordinates the concurrent execution of `LogWrite`, `AtomicWrite`, `Commit`, `Abort`, and log recovery commands across the loggers. The central controller also handles `AtomicWrite` commands. If an `AtomicWrite` specifies a write that is within a stripe of 8 kB at a single memory controller, then the central controller sends the `AtomicWrite` directly to the target logger. Otherwise, the central controller breaks the `AtomicWrite` up into the appropriate `LogWrite` commands followed by a `Commit`. In the first case, the central controller avoids the extra latency to coordinate the commit across memory controllers. In either case, system avoids an extra IO request for an explicit `Commit`.

Three hardware components work together to implement transactional operations. First, the TID manager maps virtual TIDs from application requests to physical TIDs and tracks the transaction commit sequence number for the system. Second, the transaction scoreboard tracks the state of each transaction and enforces ordering constraints during commit and recovery. Finally, the transaction status table exports a set of memory-mapped IO registers that the host system interrogates during interrupt handling to identify completed transactions.

The central controller assigns a physical TID to incoming `LogWrite` and `AtomicWrite` requests, unless they have already received a physical TID from a previous request.

To perform a `LogWrite` the central controller breaks up requests along stripe boundaries, sends local `LogWrite`s to affected memory controllers, and awaits their completion. To maximize performance, our system allows multiple `LogWrite`s from the same transaction to be in-flight at once. If the `LogWrite`s are to disjoint areas, they will behave as expected. However, if they overlap, the results are unpredictable because parts of two requests may arrive at loggers in different orders. In those cases, the application can enforce an ordering by issuing a barrier command that will force outstanding `LogWrite` requests to finish before proceeding.

On `Commit`, the central controller increments the global transaction sequence number and broadcasts a commit command with the sequence number to the memory controllers that received `LogWrite`s. The loggers respond as soon as they have completed any outstanding `LogWrite` operations and have marked the transaction as committed. When the central controller receives all responses, it signals the loggers to begin applying the log and simultaneously notifies the application that the transaction has committed. Notifying the application before the loggers have finished applying the logs hides part of the log application latency. This is safe since only a memory failure (e.g., a failing NVM memory chip) can prevent log application from eventually completing. In that case, we assume that the entire storage device has failed and the data it contains is lost (see Section 6.3).

Implementation complexity Adding support for atomic writes to the baseline system required only a modest increase in complexity and hardware resources. The Verilog implementation of the logger required 1372 lines, excluding blank lines and comments. The changes to the central controller are hard to quantify. Once placed and routed on the FPGAs, adding the eight loggers and changing the central controller increased hardware consumption by 26%.

6.3 Recovery

Our system coordinates recovery operations in the kernel driver rather than in hardware to minimize complexity. There are two problems it needs to solve: The first is that some memory controllers may have marked a transaction as COMMITTED while others have not. In this case, the transaction must abort. Second, the system must apply the transaction in the correct order (as given by their commit sequence numbers).

On boot, the driver scans the transaction tables at each memory controller to assemble a complete picture of transaction state across all the controllers. It identifies the TIDs and sequence numbers for the transactions that all loggers have marked as COMMITTED and sorts them by sequence number. The kernel then issues a kernel-only `WriteBack` command for each of these TIDs that triggers log replay at each logger. Finally, it issues `Abort` commands for all the other TIDs. Once this is complete, the array is in a consistent state, and the driver makes the array available for normal use.

6.4 Testing and verification

To verify the atomicity and durability of our multi-part atomic write interface, we added hardware support to emulate system failure and performed failure and recovery testing. This presents a challenge since the DRAM our prototype uses is volatile. To overcome this problem, we added support to force a reset of the system, which immediately suspends system activity. During system reset, we keep the memory controllers active to send refresh commands to the DRAM in order to emulate non-volatility. We assume the system includes capacitors to complete memory operations that the memory chips are in the midst of performing, just as many commercial SSDs do. To test recovery, we send a reset from the host while running a test, reboot the host system, and run our recovery protocol. Then, we run an application-specific consistency check to verify that there were no partial writes are visible.

We used two workloads during testing. The first workload consists of 16 threads each repeatedly performing an `AtomicWrite` to its own 8 kB region. Each write comprises a repeated sequence number that increments with each write. To check consistency, the application reads each of the 16 regions and verifies that they contain only a single sequence number and that that sequence number equals the last committed value. In the second workload, 16 threads continuously inserting and deleting nodes from our B+tree. After reset, reboot, and recovery, the application runs a function to verify the consistency of the B+tree.

We ran the workloads over a period of a few days, interrupting them periodically. The consistency checks for both workloads passed after every reset and recovery.

7 Results

This section measures the performance of our multi-part atomic write primitive and evaluates its impact on MARS as well as other applications that require strong consistency guarantees. We first evaluate our system through microbenchmarks that measure the basic performance characteristics. Then, we present results for MARS relative to a traditional ARIES implementation, highlighting the performance improvement in a database setting. Finally, we show results for a set of three complex persistent data structures and MemcacheDB [8], a persistent key-value store for web applications.

7.1 Latency and bandwidth

Implementing atomic writes in hardware reduces the overhead of the multi-phase write algorithms (e.g., writing a log entry and then marking it with a commit record) that applications traditionally use to write reliably to disk.

Figure 4 shows the latencies of each stage of a 512 B atomic write implemented three different ways: Using multiple, synchronous non-atomic writes ("SoftAtomic"), using `LogWrite` followed by a `Commit` ("LogWrite+Commit"), and using `AtomicWrite`. As a reference, we include the latency breakdown for a normal write. For SoftAtomic we buffer writes in memory, flush the writes to a log, write a commit record, and then write the data in place. We used a modified version of XDD [31] to collect the data.

The figure shows the transitions between hardware and software and two different latencies for each operation. The first is the *commit latency* between command initiation and when the application learns that the transaction logically commits (marked with "C"). For applications using atomic writes to implement transactions (e.g., writing to a log), the commit latency is the critical latency. The second latency, the *write back latency* is from command initiation to the completion of the write back (marked with "WB"). At this point the TID becomes available for use again.

The largest savings (41.4%) come from reducing the number of DMA transfers from three for SoftAtomic to one for the others (LogWrite+Commit takes two IO operations, but the `Commit` does not need a DMA). Using `AtomicWrite` to eliminate the separate `Commit` operation reduces latency by an additional 41.8%. Because an aligned 512 B access targets a single memory controller, the hardware can perform the atomic update at a

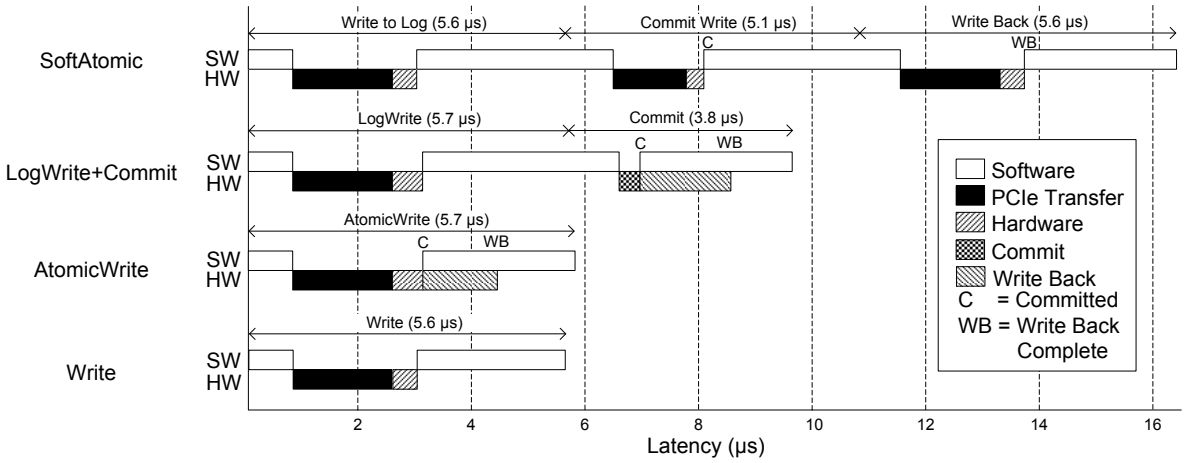


Figure 4: **Latency breakdown for 512 B atomic writes** Performing atomic writes without hardware support (top) requires three IO operations and all the attendant overheads. Using `LogWrite` and `Commit` reduces the overhead and `AtomicWrite` reduces it further by eliminating another IO operation. The latency cost of using `AtomicWrite` compared to normal writes is almost negligible.

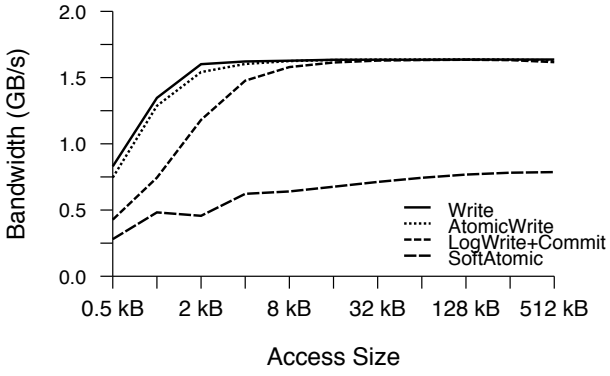


Figure 5: **Transaction throughput** By moving the log processing into the storage device, our system is able to achieve transaction throughput nearly equal to normal, non-atomic write throughput.

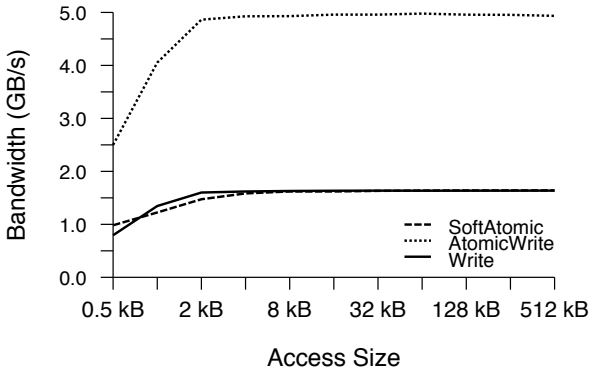
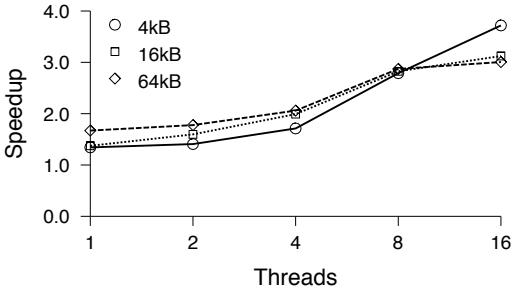


Figure 6: **Internal bandwidth** Hardware support for atomic writes allows our system to exploit the internal bandwidth of the storage array for logging and devote the PCIe link bandwidth to transferring useful data.

single logger, eliminating the coordination overhead with the central controller.

Figure 5 plots the effective bandwidth (i.e., excluding writes to the log) for atomic writes ranging in size from 512 B to 512 kB. Our scheme increases throughput by between 2 and 3.8× relative to SoftAtomic. The data also show the benefits of `AtomicWrite` for small requests: transactions smaller than 4 kB achieve 92% of the bandwidth of normal writes in the baseline system.

Figure 6 shows the source of the performance improvement for multi-part atomic writes. It plots the total bytes read or written across all the memory controllers internally. For writes, internal and external bandwidth are the same. SoftAtomic achieves the same internal bandwidth because it saturates the PCIe bus, but roughly half of that bandwidth goes to writing the log. LogWrite+Commit and `AtomicWrite` consume much more internal bandwidth (up to 5 GB/s), allowing them to saturate the PCIe link with useful data and better utilize the memory controllers.



	Threads (Swaps/sec)				
Page Size	1	2	4	8	16
4 kB	21907	40880	72677	117851	144201
16 kB	10352	19082	30596	40876	43569
64 kB	3808	5979	8282	11043	11583

Figure 7: **Comparison of MARS and ARIES** Because fast NVMs have good random write performance, there is little benefit to no-force and steal transactions. With hardware support, MARS eliminates undo logging and write backs in order to maximize bandwidth and minimize resource contention.

7.2 MARS Evaluation

This section evaluates the benefits of MARS compared to a baseline implementation of ARIES. For this experiment, our benchmark transactionally swaps objects (pages) in a large database-style table.

The baseline implementation of ARIES performs the undo and redo logging required for steal and no-force. It includes a checkpoint thread that manages a pool of dirty pages, flushing pages to the storage array as the pool fills.

Our MARS implementation uses multi-part atomic writes to eliminate no-force and steal. The hardware implements a force policy at the memory controllers and we rely on the log to hold the most recent copy of an object prior to commit, giving us the benefits of a steal policy without requiring undo logging. Using a force policy in hardware eliminates the extra IO requests needed to commit and write back data. Removing undo logging and write backs reduces the amount of data sent to the storage array over the PCIe link by a factor of two.

Figure 7 shows the throughput, measured in transactions per second, for between 1 and 16 threads concurrently swapping objects of between 4 and 64 kB. The solid lines show the performance of MARS using atomic writes and the dashed lines show the performance of the baseline implementation of ARIES. For small transactions, where logging overheads are largest, our system outperforms ARIES by as much as 3.7×. For larger objects, the gains are smaller—3.1× for 16 kB objects and 3× for 64 kB. In these cases, ARIES makes better use of the available PCIe bandwidth, compensating for some of the overhead due to additional logs writes and write backs. MARS also scales better than ARIES: We see 1.6× speedup going from 4 threads to 8 threads for 4 kB objects, compared to a 0.5% performance loss for ARIES.

7.3 Persistent data structure performance

We evaluate our system's impact on several light-weight persistent data structures designed to take advantage of our user space driver and transactional hardware support: a hash table, a B+tree, and a large scale-free graph that supports "six degrees of separation" queries.

The hash table implements a transactional key-value store. It resolves collisions using separate chaining, and it uses per-bucket locks to handle updates from concurrent threads. Typically, a transaction requires only a single write to a key-value pair. But, in some cases an update requires modifying multiple key-value pairs in a bucket's chain. The footprint of the hash table is 32 GB, and we use 25 B keys and 1024 B values for this experiment. Each thread in the workload repeatedly picks a key at random within a specified range and either inserts or removes the key-value pair depending on whether or not the key is already present.

The B+tree also implements a 32 GB transactional key-value store. It caches the index, made up of 8 kB nodes, in memory for quick retrieval. To support a high degree of concurrency, it uses Bayer and Scholnick's algorithm [1] based on node safety and lock coupling. The B+tree is a good case study for our system because transactions can be complex: An insertion or deletion may cause splitting or merging of nodes throughout the height of the tree. Each thread in this workload repeatedly inserts or deletes

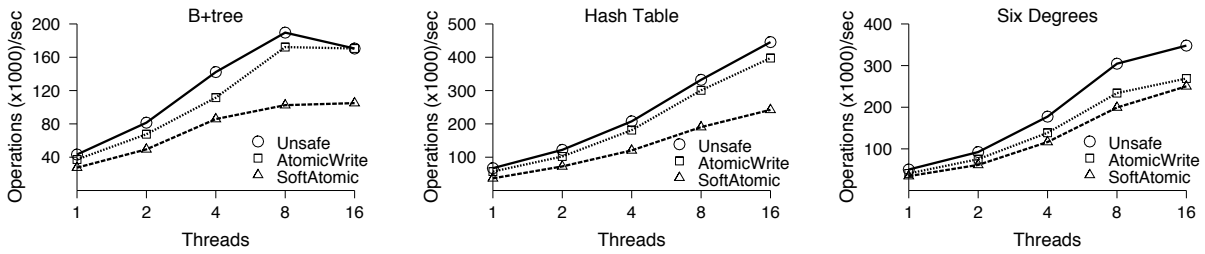


Figure 8: **Workload performance** Each set of lines compares the throughput of our (a) B+tree, (b) hash table, and (c) Six Degrees workloads for Unsafe, `AtomicWrite`, and SoftAtomic versions as we scale the number of threads.

a key-value pair at random.

Six Degrees operates on a large, scale-free graph representing a social network. It alternately performs Dijkstra's algorithm to find six-edge paths and modifies the graph by inserting or removing an edge. We use a 32 GB footprint for the undirected graph and store it in adjacency list format. Rather than storing a linked list of edges for each node, we use a linked list of edge pages, where each page contains up to 256 edges. This allows us to read many edges in a single request to the storage array. Each transactional update to the graph acquires locks on a pair of nodes and modifies each node's linked list of edges.

Figure 8 shows the performance for three implementations of each workload running with between 1 and 16 threads. The first implementation, "Unsafe," does not provide any durability or atomicity guarantees and represents an upper limit on performance. For all three workloads, adding ACID guarantees in software reduces performance by between 28 and 46% compared to Unsafe. For the B+tree and hash table, our atomic write support sacrifices just 13% of the performance of the unsafe versions on average. Six Degrees, on the other hand, sees a 21% performance drop with atomic writes because its transactions are longer and modify multiple nodes. Using atomic writes also improves scaling slightly. For instance, the `AtomicWrite` verion of HashTable closely tracks the performance improvements of the Unsafe version, with only an 11% slowdown at 16 threads while the SoftAtomic version is 46% slower.

7.4 MemcacheDB performance

To understand the impact of hardware transactional support at the application level, we integrated our hash table into MemcacheDB [8], a persistent version of Memached [20], the popular key-value store. The original Memcached uses a large hash table to store a read-only cache of objects in memory. MemcacheDB supports safe updates by using Berkeley DB to make the key-value store persistent. MemcacheDB uses a client-server architecture, and, for this experiment, we run it on a single computer acting as both clients and server.

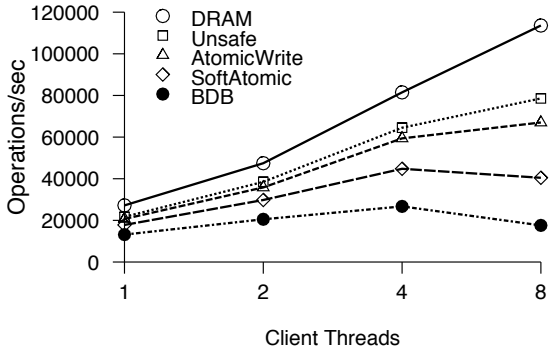


Figure 9: **MemcacheDB performance** Adding hardware support for atomicity increases performance by 1.7× for eight clients, and comes within 15% of matching the performance of an unsafe version that provides no durability.

Figure 9 compares the performance of MemcacheDB using our hash table as the key-value store (`AtomicWrite`) to versions that use volatile DRAM, a BDB database (labeled "BDB"), an in-storage key-value store without atomicity guarantees ("Unsafe"), and a SoftAtomic version. For eight threads, our system is 41% slower than DRAM and 15% slower than the Unsafe version. It is also 1.7× faster than the SoftAtomic implementation and 3.8× faster than BDB. Note that BDB provides many advanced features that add overhead but that MemcacheDB does not need and our implementation does not provide. Beyond eight threads, performance degrades because the application uses a single lock for updates.

8 Conclusion

Existing transaction mechanisms such as ARIES were designed to exploit the characteristics of disk, making them a poor fit for storage arrays of fast, non-volatile memories. We presented a redesign of ARIES, called MARS, that provides the same set of features to the application

but utilizes a novel multi-part atomic write operation that takes advantage of the parallelism and performance in fast NVM-based storage. We demonstrated MARS and multi-part atomic writes in our prototype storage array. Compared to transactions implemented in software, our system increases effective bandwidth by up to 3.8× and decreases latency by 2.9×. When applied to MARS, multi-part atomic writes yield a 3.7× performance improvement relative to a baseline implementation of ARIES requiring both redo and undo logging of pages. Across a range of persistent data structures, multi-part atomic writes improve operation throughput by an average of 1.4×.

References

[1] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[2] http://beecube.com/products/.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[4] M. J. Breitwisch. Phase change memory. *Interconnect Technology Conference, 2008. IITC 2008. International*, pages 219–221, June 2008.

[5] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 43, pages 385–395, New York, NY, USA, 2010. ACM.

[6] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 387–400, New York, NY, USA, 2012. ACM.

[7] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9R1, HP Laboratories, November 1992.

[8] S. Chu. Memcachedb. http://memcachedb.org/.

[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 *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 105–118, New York, NY, USA, 2011. ACM.

[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 *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[11] O. Corporation. ZFS. http://hub.opensolaris.org/bin/view/Community+Group+zfs/.

[12] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: a new approach to improving file systems. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 15–28, New York, NY, USA, 1993. ACM.

[13] B. Dieny, R. Sousa, G. Prenat, and U. Ebels. Spin-dependent phenomena and their implementation in spintronic devices. *VLSI Technology, Systems and Applications, 2008. VLSI-TSA 2008. International Symposium on*, pages 70–71, April 2008.

[14] R. Grimm, W. Hsieh, M. Kaashoek, and W. de Jonge. Atomic recovery units: failure atomicity for logical disks. *Distributed Computing Systems, International Conference on*, 0:26–37, 1996.

[15] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter*, pages 235–246, 1994.

[16] International technology roadmap for semiconductors: Emerging research devices, 2009.

[17] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *Proc. VLDB Endow.*, 3:681–692, September 2010.

[18] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 2009. ACM.

[19] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 92–101, New York, NY, USA, 1997. ACM.

[20] Memcached. http://memcached.org/.

[21] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[22] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *High Performance Computer*

Architecture (HPCA), 2011 IEEE 17th International Symposium on, pages 301–311, February 2011.

[23] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 147–160, Berkeley, CA, USA, 2008. USENIX Association.

[24] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[25] M. K. Qureshi, A. Seznec, L. A. Lastras, and M. M. Franceschini. Practical and secure PCM systems by online detection of malicious write streams. *High-Performance Computer Architecture, International Symposium on*, 0:478–489, 2011.

[26] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 146–160, New York, NY, USA, 1993. ACM.

[27] S. Schechter, G. H. Loh, K. Straus, and D. Burger. Use ecp, not ecc, for hard failures in resistive memories. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 141–152, New York, NY, USA, 2010. ACM.

[28] R. Sears and E. Brewer. Stasis: flexible transactional storage. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 29–44, Berkeley, CA, USA, 2006. USENIX Association.

[29] R. Sears and E. Brewer. Segment-based recovery: write-ahead logging revisited. *Proc. VLDB Endow.*, 2:490–501, August 2009.

[30] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS '11: Proceeding of the 16th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, 2011. ACM.

[31] XDD version 6.5. http://www.ioperformance.com/.