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

Joel Coburn* Trevor Bunker* Meir Schwarz Rajesh Gupta Steven Swanson
Department of Computer Science and Engineering
University of California, San Diego
{jdcoburn,tbunker,rgupta,swanson}@cs.ucsd.edu

Abstract

Transaction-based systems often rely on write-ahead logging (WAL) algorithms designed to maximize performance on disk-based storage. However, emerging fast, byte-addressable, non-volatile memory (NVM) technologies (e.g., phase-change memories, spin-transfer torque MRAMs, and the memristor) present very different performance characteristics, so blithely applying existing algorithms can lead to disappointing performance.

This paper presents a novel storage primitive, called editable atomic writes (EAW), that enables sophisticated, highly-optimized WAL schemes in fast NVM-based storage systems. EAWs allow applications to safely access and modify log contents rather than treating the log as an append-only, write-only data structure, and we demonstrate that this can make implementating complex transactions simpler and more efficient. We use EAWs to build MARS, a WAL scheme that provides the same as features ARIES [26] (a widely-used WAL system for databases) but avoids making disk-centric implementation decisions.

We have implemented EAWs and MARS in a next-generation SSD to demonstrate that the overhead of EAWs is minimal compared to normal writes, and that they provide large speedups for transactional updates to hash tables, B+trees, and large graphs. In addition, MARS outperforms ARIES by up to 3.7× while reducing software complexity.

*Now working at Google Inc.

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522724

1 Introduction

Emerging fast non-volatile memory (NVM) technologies, such as phase change memory, spin-torque transfer memory, and the memristor promise to be 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 performance by exploiting the speed of these memories. Recent work focuses on optimizing read and write performance in these systems [6, 7, 41]. But these new memory technologies also enable new approaches to ensuring data integrity in the face of failures.

File systems, databases, persistent object stores, and other applications rely on strong consistency guarantees for persistent data structures. 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 use sophisticated, disk-based optimizations to minimize the cost of synchronous writes and leverage the sequential bandwidth of disk. For example, WAL-based systems write data to a log before updating the data in-place, but they typically delay the in-place updates until they can be batched into larger sequential writes.

NVM technologies provide very different performance characteristics, and exploiting them requires new approaches to providing 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.

This paper presents a novel WAL scheme, called *Modified ARIES Redesigned for SSDs (MARS)*, optimized for NVM-based storage. The design of MARS reflects an examination of ARIES [26], a popular WAL-based recovery algorithm for databases, in the context of these new memories. MARS provides the same high-level features for implementing effi-

cient and robust transactions as ARIES, but without any of the disk-based design decisions ARIES incorporates.

To support MARS, we designed a novel multi-part atomic write primitive, called editable atomic writes (EAW). An EAW consists of a set of redo log entries, one per object to be modified, that the application can freely update multiple times in-place in the log prior to commit. Once committed, the *SSD hardware copies the final values from the log to their target locations*, and the copy is guaranteed to succeed even in the presence of power or host system failures. EAWs make implementing ARIES-style transactions simpler and faster, giving rise to MARS. EAWs are also a useful building block for other applications that must provide strong consistency guarantees.

The EAW 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 safely exposes the logs to the application and allows it to manage the log space directly, providing the flexibility that sophisticated WAL schemes like MARS require. In contrast, recent work on atomic write support for flash-based SSDs [29, 30] hides the logging in the flash translation layer (FTL). Consequently, ARIES-style logging schemes must write data to both a software-accessible log and to its final destination, resulting in higher bandwidth consumption and lower performance.

We implemented EAWs in a prototype PCIe-based storage array [6]. Microbenchmarks show that they reduce latency by 2.9× compared to using normal synchronous writes to implement a traditional WAL protocol, and EAWs increase effective bandwidth by between 2.0 and 3.8× by eliminating logging overheads. Compared to non-atomic writes, EAWs reduce effective bandwidth just 1-8% and increase latency by just 30%.

We use EAWs to implement MARS, to implement simple on-disk persistent data structures, and to modify MemcacheDB [10], a persistent version of memcached. MARS improves performance by 3.7× relative to our baseline version of ARIES. EAWs speed up our ACID key-value stores based on a hash table and a B+tree by 1.5× and 1.4×, respectively, relative to a software-based version, and EAWs improve performance for a simple online scale-free graph query benchmark by 1.3×. Furthermore, performance for EAW-based versions of these data structures is only 15% slower than non-transactional versions. For MemcacheDB, replacing Berkeley DB with an EAW-based key-value store improves 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 ARIES in the context of fast NVM-based storage and describes EAWs and MARS. Section 4 describes our implementation of EAWs in hardware. Section 5 evaluates EAWs and their impact on the performance of MARS and other persistent data structures. Section 6 places this work in the context of prior work on support for transactional storage. In Section 7, we discuss the limitations of our approach and some areas for future work. Section 8 summarizes our contributions.

2 Storage technology

Fast NVMs will catalyze changes in the organization of storage arrays and in how applications and the OS access and manage storage. This section describes the architecture of the storage system that our work targets and the memory technologies it uses. Section 4 describes our implementation in more detail.

Fast non-volatile memories such as phase-change memories (PCM) [3], spin-transfer torque [15] memories, and the memristor differ fundamentally from conventional disks and from the flash-based SSDs that are beginning to replace them. The most important features of NVMs are their performance (relative to disk and flash) and their simpler interface (relative to flash).

Predictions by industry [19] and academia [3, 15] 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. In addition, technologies such as PCM will have a significant density and cost-per-bit advantage (estimated 2 to 4×) over DRAM [3, 33].

These device characteristics require storage architectures with topologies and interconnects capable of exploiting their low latency and high bandwidth.

Modern high-end SSDs often attach to the host via PCIe, and this approach will work well for fast NVMs too. PCIe bandwidth is scalable and many high end processors have nearly as much PCIe bandwidth as main memory bandwidth. PCIe also offers scalable capacity since any number and type of memory channels (and memory controllers) can sit behind a PCIe endpoint. This makes a PCIe-attached architecture a natural candidate for capacity-intensive applications like databases, graph analytics, or caching. Multiple hosts can also connect to a single device over PCIe, allowing for increased availability if one host fails. Finally, the appearance of NVMExpress [27]-based SSDs (e.g., Intel's Chatham NVMe drive [11]) signals that PCIe-attached SSDs are a likely target design for fast NVMs in the near term.

A consequence of PCIe SSDs' scalable capacity is that their internal memory bandwidth will often exceed their PCIe bandwidth (e.g., 8:1 ratio in our prototype SSD). Unlike flash memory where many chips can hang off a single bus, the low latency of fast NVMs requires minimal loading on data buses connecting chips and memory controllers. As a result, large capacity devices must spread storage across many memory channels. This presents an opportunity to exploit this surfeit of internal bandwidth by offloading tasks to the storage device.

Alternately, fast NVMs can attach directly to the proces-

sor's memory bus, providing the lowest latency path to storage and a simple, memory-like interface. However, reduced latency comes at a cost of reduced capacity and availability since the pin, power, and signaling constraints of a computer's memory channels will limit capacity and fail-over will be impossible.

In this work, we focus on PCIe-attached storage architectures. Our baseline storage array is the Moneta-Direct SSD [6, 7], which spreads 64 GB of DRAM 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 [23] (48 ns and 150 ns for reads and writes, respectively, to the memory array inside the memory chips). The techniques we describe would also work in STT-MRAM or memristor-based systems.

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 [31, 32, 35]. Moneta uses start-gap wear leveling [31]. With fast, in-place updates, Moneta is able to provide low-latency, high-bandwidth access to storage that is limited only by the PCIe interconnect between the host and the device.

3 Complex Transaction Support in Fast SSDs

The design of ARIES and other data management systems (e.g., journaling file systems) relies critically on the atomicity, durability, and performance properties of the underlying storage hardware. Data management systems combine these properties with locking protocols, rules governing the order of updates, and other invariants to provide application-level transactional guarantees. As a result, the semantics and performance characteristics of the storage hardware play a key role in determining the implementation complexity and overall performance of the complete system.

We have designed a novel multi-part atomic write primitive, called editable atomic writes (EAW), that supports complex logging protocols like ARIES-style write-ahead logging. In particular, EAWs make it easy to support transaction isolation in a scalable way while aggressively leveraging the performance of next-generation, non-volatile memories. This feature is missing from existing atomic write interfaces [29, 30] designed to accelerate simpler transaction models (e.g., file metadata updates in journaling file systems) on flash-based SSDs.

This section describes EAWs, presents our analysis of ARIES and the assumptions it makes about disk-based storage, and describes MARS, our reengineered version of ARIES that uses EAWs to simplify transaction processing and improve performance.

3.1 Editable atomic writes

The performance characteristics of disks and, more recently, SSDs have deeply influenced most WAL schemes. Since sequential writes are much faster than random writes, WAL schemes maintain their logs as append-only sets of records, avoiding long-latency seeks on disks and write amplification in flash-based SSDs. However, for fast NVMs, there is little performance difference between random and sequential writes, so the advantages of an append-only log are less clear.

In fact, append- and write-only logs add complexity because systems must construct and maintain in-memory copies that reflect the operations recorded in the log. For large database transactions the in-memory copies can exceed available memory, forcing the database to spill this data onto disk. Consequently, the system may have three copies of the data at one time: one in the log, the spilled copy, and the data itself. However, if the log data resides in a fast NVM storage system, spilling is not necessary – the updated version of the data resides in the log and the system reads or updates it without interfering with other writes to the log. Realizing this capability requires a new flexible logging primitive which we call editable atomic writes (EAW).

EAWs use write-ahead redo logs to combine multiple writes to arbitrary storage locations into a single atomic operation. EAWs make it easy for applications to provide isolation between transactions by keeping the updates in a log until the atomic operation commits and exposing that log to the application so that a transaction can see its own updates and freely update that data multiple times prior to commit. EAWs are simple to use and strike a balance between implementation complexity and functionality while allowing our SSD to leverage the performance of fast NVMs.

EAWs require the application to allocate space for the log (e.g., by creating a log file) and to specify where the redo log entry for each write will reside. This avoids the need to statically allocate space for log storage and ensures that the application knows where the log entry resides so it can modify it as needed.

The implementation of EAWs is spread across the storage device hardware and system software. Hardware support in the SSD is responsible for logging and copying data to guarantee atomicity. Applications use the EAW library interface to perform common IO operations by communicating di-

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.
`AtomicWrite`(TID, file, offset, data, len, logfile, logoffset)	Create and commit a transaction containing a single write.
`NestedTopAction`(TID, logfile, logoffset)	Commit a nested top action by applying the log from a specified starting point to the end.
`Abort`(TID) `PartialAbort`(TID, logfile, logoffset)	Cancel the transaction entirely, or perform a partial rollback to a specified point in the log.

Table 1: EAW commands. These commands perform multi-part atomic updates to the storage array and help support user-level transactions.

rectly with hardware, but storage management policy decisions still reside in the kernel and file system.

Below, we describe how an application initiates an EAW, commits it, and manages log storage in the device. We also discuss how the EAW interface makes transactions robust and efficient by supporting partial rollbacks and nested top actions. Then, we outline how EAWs help simplify and accelerate ARIES-style transactions. In Sections 4 and 5 we show that the hardware required to implement EAWs is modest and that it can deliver large performance gains.

Creating transactions Applications execute EAWs using the commands in Table 1. Each application accessing the storage device has a private set of 64 transaction IDs (TIDs)[1], and the application is responsible for tracking which TIDs are in use. TIDs can be in one of three states: FREE (the TID is not in use), PENDING (the transaction is underway), or COMMITTED (the transaction has committed). TIDs move from COMMITTED to FREE when the storage system notifies the host that the transaction is complete.

To create a new transaction with TID, T, the application passes T to `LogWrite` along with information that specifies the data to write, the ultimate target location for the write (i.e., a file descriptor and offset), and the location for the log data (i.e., a log file descriptor and offset). This operation copies the write data to the log file but does not modify the target file. After the first `LogWrite`, the state of the transaction changes from FREE to PENDING. 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, a transaction can read its own data prior to commit by explicitly reading from the locations in the log containing that data. After an initial `LogWrite` to a storage location, a transaction may update that data again before commit by issuing a (non-logging) write to the corresponding log location.

Committing transactions The application commits the transaction with `Commit`(T). In response, the storage array assigns the transaction a commit sequence number that determines the commit order of this transaction relative to others. It then atomically applies the `LogWrite`s by copying the data from the log to their target locations.

When the `Commit` command completes, the transaction has logically committed, and the transaction moves to the COMMITTED state. If a system failure should occur after a transaction logically commits but before the system finishes writing the data back, then the SSD will replay the log during recovery to roll the changes forward.

When log application completes, 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 and reuse the TID.

Robust and efficient execution The EAW interface provides four other commands designed to make transactions robust and efficient through finer-grained control over their execution. The `AtomicWrite` command creates and commits a single atomic write operation, saving one IO operation for singleton transactions. The `NestedTopAction` command is similar to `Commit` but instead applies the log from a specified offset up through the tail and allows the transaction to continue afterwards. This is useful for operations that should commit independent of whether or not the transaction commits (e.g., extending a file or splitting a page in a B-tree), and it is critical to database performance under high concurrency.

Consider an insert of a key into a B-tree where a page split must occur to make room for the key. Other concurrent insert operations might either cause an abort or be aborted themselves, leading to repeatedly starting and aborting a page split. With a `NestedTopAction`, the page split occurs once and the new page is immediately available to other transactions.

The `Abort` command aborts a transaction, releasing all resources associated with it, allowing the application

[1]This is not a fundamental limitation but rather an implementation detail of our prototype. Supporting more concurrent transactions increases resource demands in the FPGA implementation. A custom ASIC implementation (quickly becoming commonplace in high-end SSDs) could easily support 100s of concurrent transactions.

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

Table 2: ARIES features. Regardless of the storage technology, ARIES must provide these features to the rest of the system.

to resolve conflicts and ensure forward progress. The `PartialAbort` command truncates the log at a specified location, or savepoint [16], to support partial rollback. Partial rollbacks are a useful feature for handling database integrity constraint violations and resolving deadlocks [26]. They minimize the number of repeated writes a transaction must perform when it encounters a conflict and must restart.

Storing the log The system stores the log in a pair of ordinary files in the storage device: a *log file* and a *log metadata file*. The log file holds the data for the log. The application creates the log file just like any other file and is responsible for allocating parts of it to `LogWrite` operations. The application can access and modify its contents at any time.

The log metadata file contains information about the target location and log data location for each `LogWrite`. The contents of the log metadata file are privileged, since it contains raw storage addresses rather than file descriptors and offsets. Raw addresses are necessary during crash recovery when file descriptors are meaningless and the file system may be unavailable (e.g., if the file system itself uses the EAW interface). A system daemon, called the *metadata handler*, "installs" log metadata files on behalf of applications and marks them as unreadable and immutable from software. Section 4 describes the structure of the log metadata file in more detail.

Conventional storage 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.

3.2 Deconstructing ARIES

The ARIES [26] framework for write-ahead logging and recovery has influenced the design of many commercial databases and acts as a key building block in providing fast, flexible, and efficient ACID transactions. Our goal is to build a WAL scheme that provides all of ARIES' features but without the disk-centric design decisions.

At a high level, ARIES operates as follows. Before modifying an object (e.g., a row of a table) in storage, ARIES records the changes in a persistent log. To make recovery robust and to allow disk-based optimizations, ARIES records both the old version (undo) and new version (redo) of the data. ARIES can only update the data in-place after the log reaches storage. On restart after a crash, ARIES brings the system back to the exact state it was in before the crash by applying the redo log. Then, ARIES reverts the effects of any uncommitted transactions active at the time of the crash by applying the undo log, thus bringing the system back to a consistent state.

ARIES has two primary goals: First, it aims to provide a rich interface for executing scalable, ACID transactions. Second, it aims to maximize performance on disk-based storage systems.

ARIES achieves the first goal by providing several important *features* to higher-level software (e.g., the rest of the database), listed in Table 2, that make it useful to a variety of applications. For example, ARIES offers flexible storage management by supporting 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 portions 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 achieve high performance on disk-based systems, ARIES incorporates a set of *design decisions* (Table 3) that exploit the properties of disk: ARIES optimizes for long, sequential accesses and avoids short, random accesses whenever possible. These design decisions are a poor fit for fast NVMs that provide fast random access, abundant internal bandwidth, and ample parallelism. Below, we describe the design decisions ARIES makes that optimize for disk and 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 (a sequential write) before it updates the object itself (a random write). To keep random writes off the critical path, ARIES uses a *no-force* policy that writes updated pages back to disk lazily after commit. In fast NVM-based storage, random writes are no more expensive than sequential writes, so the value of no-force is much lower.

Steal ARIES uses a *steal* policy to allow the buffer manager to "page out" uncommitted, dirty pages to disk during transaction execution. This lets the buffer manager support transactions larger than the buffer pool, group writes together to take advantage of sequential disk bandwidth, and avoid data races on pages shared by overlapping transactions. However, stealing requires undo logging so the system can roll back the uncommitted changes if the transaction aborts.

As a result, ARIES writes both an undo log *and* a redo log to disk in addition to eventually writing back the data in

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	ARIES performs updates on pages Page writes are atomic	Hardware uses pages Software operates on objects
Log Sequence Numbers (LSNs)	Simplify recovery Enable high-level features	ARIES orders updates to storage using LSNs	Hardware enforces ordering with commit sequence numbers

Table 3: ARIES design decisions. ARIES relies on a set disk-centric optimizations to maximize performance on disk-based storage. However, these optimizations are a poor fit for fast NVM-based storage, and we present alternatives to them in MARS.

place. This means that, roughly speaking, writing one logical byte to the database requires writing three bytes to storage. For disks, this is a reasonable tradeoff because it avoids placing random disk accesses on the critical path and gives the buffer manager enormous flexibility in scheduling the random disk accesses that must occur. For fast NVMs, however, random and sequential access performance are nearly identical, so this trade-off needs to be re-examined.

Pages and Log Sequence Numbers (LSNs) ARIES uses disk pages as the basic unit of data management and recovery and uses the atomicity of page writes as a foundation for larger atomic writes. This reflects the inherently block-oriented interface that disks provide. ARIES also embeds a log sequence number (LSN) in each page to establish an ordering on updates and determine how to reapply them during recovery.

As recent work [37] highlights, pages and LSNs complicate several aspects of database design. Pages make it difficult to manage objects that span multiple pages or are smaller than a single page. Generating globally unique LSNs limits concurrency and embedding LSNs in pages complicates reading and writing objects that span multiple pages. LSNs also effectively prohibit simultaneously writing multiple log entries.

Advanced NVM-based storage arrays that implement EAWs can avoid these problems. The hardware motivation for page-based management does not exist for fast NVMs, so EAWs expose a byte-addressable interface with a much more flexible notion of atomicity. Instead of an append-only log, EAWs provide a log that can be read and written throughout the life of a transaction. This means that undo logging is unnecessary because data will never be written back in-place before a transaction commits. Also, EAWs implement ordering and recovery in the storage array itself, eliminating the need for application-visible LSNs.

3.3 Building MARS

MARS is an alternative to ARIES that implements the same features but reconsiders ARIES' design decisions in the context of fast NVMs and EAW operations. Like ARIES, MARS plays the role of the logging component of a database storage manager such as Shore [5, 20].

MARS differs from ARIES in three key ways. First, MARS relies on the storage device, via EAW operations, to apply the redo log at commit time. Second, MARS eliminates the undo log that ARIES uses to implement its page stealing mechanism but retains the benefits of stealing by relying on the editable nature of EAWs. Third, MARS abandons the notion of transactional pages and instead operates directly on objects while relying on the hardware to guarantee ordering.

MARS uses `LogWrite` operations for transactional updates to objects (e.g., rows of a table) in the database. This provides several advantages. Since `LogWrite` does not update the data in-place, the changes are not visible to other transactions until commit. This makes it easy for the database to implement isolation. MARS also uses `Commit` to efficiently apply the log.

This change means that MARS "forces" updates to storage on commit, unlike the no-force policy traditionally used by ARIES. `Commit` executes entirely within the SSD, so it can utilize the full internal memory bandwidth of the SSD (32 GB/s in our prototype) to apply the commits and avoid consuming IO interconnect bandwidth and CPU resources.

When a large transaction cannot fit in memory, ARIES can safely page out uncommitted data to the database tables because it maintains an undo log. MARS has no undo log but must still be able to page out uncommitted state. Instead of writing uncommitted data to disk at its target location, MARS writes the uncommitted data directly to the redo log entry corresponding to the `LogWrite` for that location. When the system issues a `Commit` for the transaction, the SSD will write the updated data into place.

By making the redo log editable, the database can use the log to hold uncommitted state and update it as needed. In MARS, this is critical in supporting robust and complex transactions. Transactions may need to update the same data multiple times and they should see their own updates prior to

commit. This is a behavior we observed in common workloads such as the OLTP TPC-C benchmark.

Finally, MARS operates on arbitrary-sized objects directly rather than on pages and avoids using LSNs by relying on the commit ordering that EAWs provide.

Combining these optimizations eliminates the disk-centric overheads that ARIES incurs and exploits the performance of fast NVMs. MARS eliminates the data transfer overhead in ARIES: MARS sends one byte over the storage interconnect for each logical byte the application writes to the database. MARS also leverages the bandwidth of the NVMs inside the SSD to improve commit performance. Section 5 quantifies these advantages in detail.

Through EAWs, MARS provides many of the features that ARIES exports to higher-level software while reducing software complexity. MARS supports flexible storage management and fine-grained locking with minimal overhead by using EAWs. They provide an interface to directly update arbitrary-sized objects, thereby eliminating pages and LSNs.

The EAW interface and hardware supports partial rollbacks via savepoints by truncating the log to a user-specified point, and MARS needs partial rollbacks to recover from database integrity constraint violations and maximize performance under heavy conflicts. The hardware also supports nested top actions by committing a portion of the log within a transaction before it completes execution. This enables high concurrency updates of data structures such as B-trees.

MARS does not currently support operation logging with EAWs. Unless the hardware supports arbitrary operations on stored data, the play back of a logged operation must trigger a callback to software. It is not clear if this would provide a performance advantage over our hardware-accelerated value logging, and we leave it as a topic for future work. MARS provides recovery independence through a flexible recovery algorithm that can recover select areas of memory by replaying only the corresponding log entries and bypassing those pertaining to failed memory cells.

4 Implementation

In this section, we present the details of the implementation of the EAW interface MARS relies on. This includes how applications issue commands to the array, how software 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.

4.1 Software support

To make logging transparent and flexible, we leverage the existing software stack of the Moneta-Direct SSD by extending the user-space driver to implement the EAW API. In addition, we utilize the XFS [38] file system to manage the logs, making them accessible through an interface that lets the user control the layout of the log in storage.

User-space driver The Moneta-Direct SSD provides a highly-optimized (and unconventional) interface for accessing data [7]. It provides a user-space driver that allows each 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 the SSD, 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. The user-space interface lets our SSD perform IO operations very quickly: 4 kB reads and writes execute in $\sim$7 μs. Our system uses this user-space interface to issue `LogWrite`, `Commit`, `Abort`, `AtomicWrite`, and other requests to the storage array.

Storing logs in the file system Our system stores the log and metadata files that EAWs require in the file system.

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 as needed, based on the application's log space requirements, using normal file IO.

The metadata file records information about each update including the target location for the redo data upon transaction commit. The metadata file is analogous to a file's inode in that it acts as a log's index into the storage array. A trusted process called the *metadata handler* creates and manages a metadata file on the application's behalf.

The system protects the metadata file 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.

To take advantage of the parallelism and internal bandwidth of the SSD, the user-space driver ensures the data offset and log offset for `LogWrite` and `AtomicWrite` requests target the same memory controller in the storage array. We make this guarantee by allocating space in extents aligned to and in multiples of the SSD's 64 kB stripe width. With XFS, we achieve this by setting the stripe unit parameter with `mkfs.xfs`.

4.2 Hardware support

The implementation of EAWs divides functionality between two types of hardware components. The first is a logging module, called the *logger*, that resides at each of Moneta's eight memory controllers and handles logging for the local controller (the gray boxes to the right of the dashed line in Figure 1). The second is a set of modifications to the central controller (the gray boxes to the left of the dashed line in Figure 1) that orchestrates operations across the eight loggers. Below, we describe the layout of the log and the components and protocols the system uses to coordinate logging, commit, and recovery.

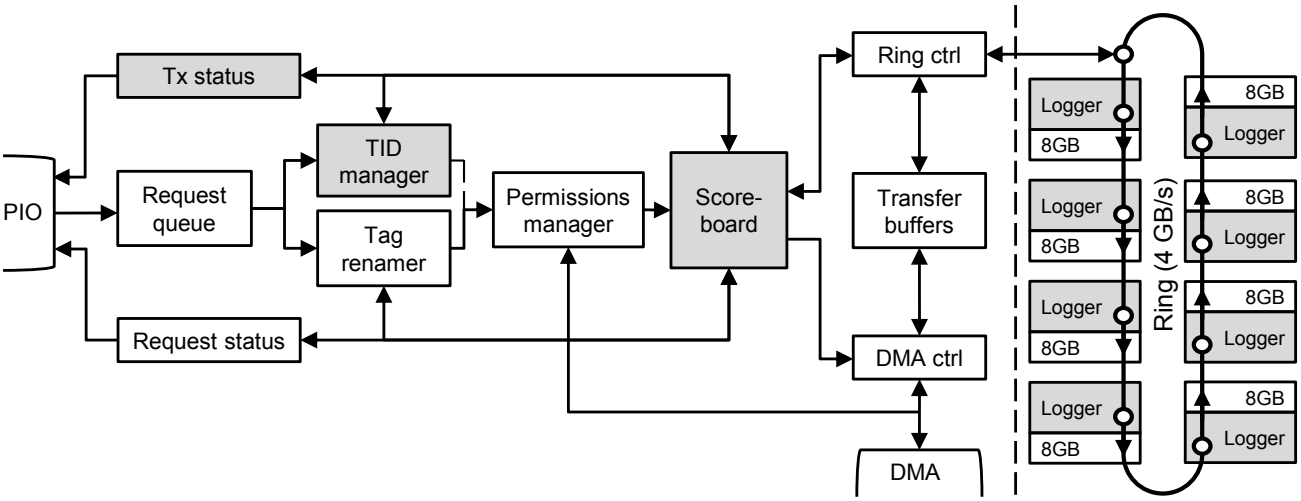


Figure 1: SSD controller architecture. To support EAWs, our we added hardware support (gray boxes) to an existing prototype SSD. The main controller (to the left of the dotted line) manages transaction IDs and uses a scoreboard to track the status of in-flight transactions. Eight loggers perform distributed logging, commit, and recovery at each memory controller.

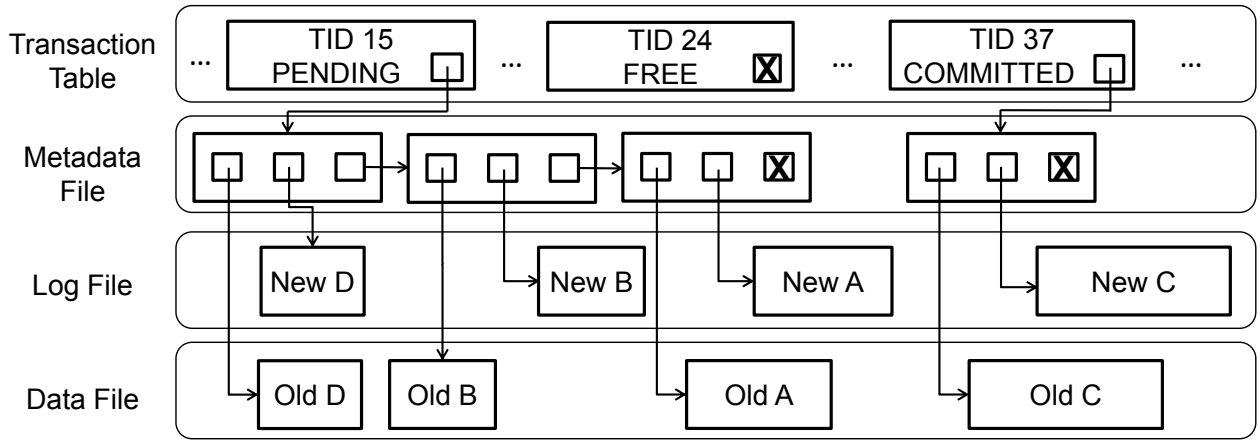


Figure 2: Example log layout at a logger. Each logger maintains a log for up to 64 TIDs. The log is a linked list of metadata entries with a transaction table entry pointing to the head of the list. The transaction table entry maintains the state of the transaction and the metadata entries contain information about each `LogWrite` request. Each link in the list describes the actions for a `LogWrite` that will occur at commit.

Logger Each logger module independently performs logging, commit, and recovery operations and handles accesses to the 8 GB of NVM storage at the memory controller. Figure 2 shows how each logger independently maintains a per-TID log as a collection of three types of entries: transaction table entries, metadata entries, and log file entries.

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

When the metadata handler installs a metadata file, the hardware divides it into fixed-size *metadata entries*. Each metadata entry contains information about a log file entry and the address of the next metadata entry for the same transaction. The log file entry contains the redo data that the logger will write back when the transaction commits.

The log for a particular TID at a logger is a linked list of metadata entries with the transaction table entry pointing to the head of the list. The complete log for a given TID (across the entire storage device) is simply the union of each logger's log for that TID.

Figure 2 illustrates the state for three TIDs (15, 24, and 37) at one logger. In this example, the application has performed three `LogWrite` requests for TID 15. For each request, the logger allocated a metadata entry, copied the data to a location in the log file, recorded the request information in the metadata entry, and then appended the metadata entry to the log. The TID is still in a PENDING state until the application issues a `Commit` or `Abort` request.

The application sends an `Abort` request for TID 24. The logger then deallocates all assigned metadata entries and clears the transaction status returning it to the FREE state.

When the application issues a `Commit` request for TID 37, the logger waits for all outstanding writes to the log file to complete and then marks the transaction as COMMITTED.

After the loggers transition to the COMMITTED state, the central controller (see below) directs each logger to apply their respective log. To apply the log, the logger reads each metadata entry in the log linked list, copying the redo data from the log file entry to its destination address. During log application, the logger suspends other read and write operations to the storage it manages to make the updates atomic. At the end of log application, the logger deallocates the transaction's metadata entries and returns the TID to the FREE state.

The central controller A single transaction may require the coordinate efforts of one or more loggers. The central controller (the left hand portion of Figure 1) coordinates the concurrent execution of the EAW commands and log recovery commands across the loggers.

Three hardware components work together to implement transactional operations. First, the TID manager maps virtual TIDs from application requests to physical TIDs and assigns each transaction a commit sequence number. 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.

To perform a `LogWrite` the central controller breaks up requests along stripe boundaries, sends local `LogWrites` to the affected loggers, and awaits their completion. To maximize performance, our system allows multiple `LogWrites` from the same transaction to be in-flight at once. If the `LogWrites` are to disjoint areas of the log, then 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. The application is responsible for ensuring that `LogWrites` do not conflict by issuing a barrier or waiting for the completion of an outstanding `LogWrite`.

The central controller implements a two-phase commit protocol. The first phase begins when the central controller receives a `Commit` command from the host. It increments the global transaction sequence number and broadcasts a commit command with the sequence number to the loggers that received `LogWrites` for that transaction. The loggers respond as soon as they have completed any outstanding `LogWrite` operations and have marked the transaction COMMITTED. The central controller moves to the second phase after it receives all the 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 and allows the application to release locks sooner so other transactions may read or write the data. This is safe since only a memory failure (e.g., a failing NVM memory chip) can prevent log application from eventually completing and the log application is guaranteed to complete before subsequent operations. In the case of a memory failure, we assume that the entire storage device has failed and the data it contains is lost (see Section 4.3).

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 harder to quantify, but were small relative to the existing central controller code base.

4.3 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: First, some loggers may have marked a transaction as COMMITTED while others have not, meaning that the data was only partially written to the log. In this case, the transaction must abort.

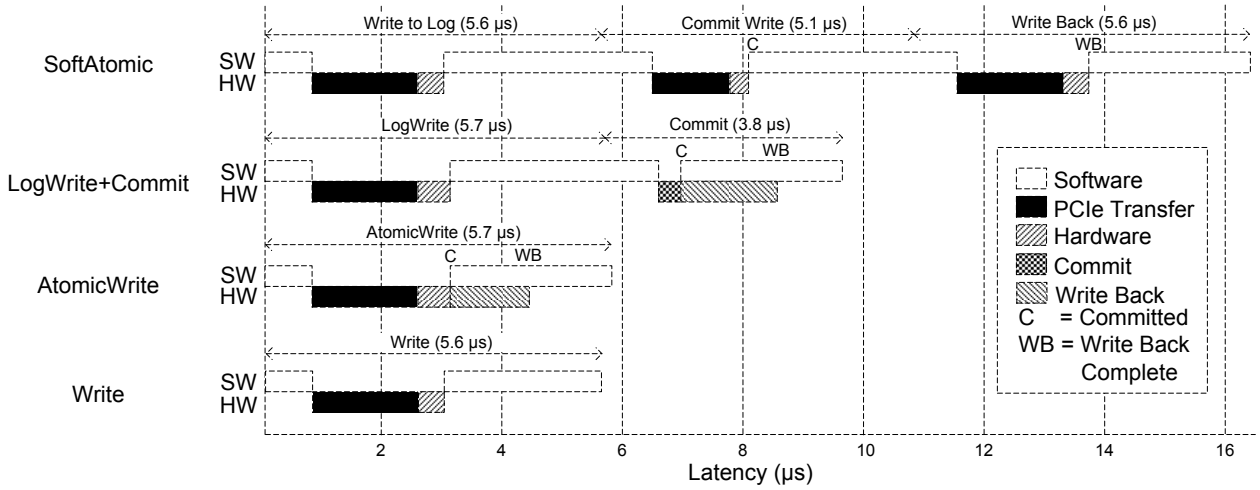


Figure 3: Latency breakdown for 512 B atomic writes. Performing atomic writes without hardware support (top) requires three IO operations and all the attendant overheads. Using `LogWrite` and `Commit` reduces the overhead and `AtomicWrite` reduces it further by eliminating another IO operation. The latency cost of using `AtomicWrite` compared to normal writes is very small.

Second, the system must apply the transactions in the correct order (as given by their commit sequence numbers).

On boot, the driver scans the transaction tables of each logger to assemble a complete picture of transaction state across all the controllers. Each transaction either completed the first phase of commit and rolls forward or it failed and gets canceled. The driver 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.

4.4 Testing and verification

To verify the atomicity and durability of our EAW implementation, 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 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 consists of 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 insert and delete nodes from our B+tree. After reset, reboot, and recovery, the application runs a consistency check 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.

5 Results

This section measures the performance of EAWs and evaluates their 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 benefits of EAWs for databases. Finally, we show results for three persistent data structures and MemcacheDB [10], a persistent key-value store for web applications.

5.1 Latency and bandwidth

EAWs eliminate the overhead of multiple writes that systems traditionally use to provide atomic, consistent updates to storage. Figure 3 measures that overhead for each stage of a 512 B atomic write. The figure shows the overheads for a traditional implementation that uses multiple synchronous non-atomic writes (“SoftAtomic”), an implementation that uses `LogWrite` followed by a `Commit` (“LogWrite+Commit”), and one that uses `AtomicWrite`. As

a reference, we include the latency breakdown for a single non-atomic write as well. 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 [40] 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 this is the critical latency, since it corresponds to the write logically completing. 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 system has finished updating data in place and the TID becomes available for use again.

The largest source of latency reduction (accounting for 41.4%) comes 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%.

Figure 4 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.

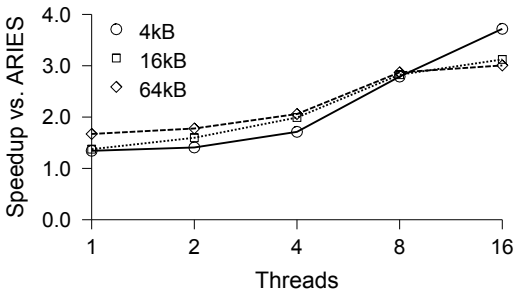
Figure 5 shows the source of the bandwidth performance improvement for EAWs. It plots the total bytes read or written across all the memory controllers internally. For normal writes, internal and external bandwidth are the same. SoftAtomic achieves the same internal bandwidth because it saturates the 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 to confine logging operations to the storage device where they can leverage the internal memory controller bandwidth.

5.2 MARS Evaluation

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

The baseline implementation of ARIES follows the description in [26] very closely and runs on the Moneta-Direct SSD. It performs the undo and redo logging required for steal and no-force and includes a checkpoint thread that manages a pool of dirty pages, flushing pages to the storage array as the pool fills.

Figure 6 shows the speedup of MARS compared to ARIES for between 1 and 16 threads running a simple microbenchmark. Each thread selects two aligned blocks of



	Swaps/sec for MARS by Thread Count				
Size	1	2	4	8	16
4 kB	21,907	40,880	72,677	117,851	144,201
16 kB	10,352	19,082	30,596	40,876	43,569
64 kB	3,808	5,979	8,282	11,043	11,583

Figure 6: Comparison of MARS and ARIES. Designing MARS to take advantage of fast NVMs allows it to scale better and achieve better overall performance than ARIES.

data between 4 and 16 kB (x-axis) at random and swaps their contents using a MARS or ARIES-style transaction. 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 log writes and write backs. MARS scales better than ARIES: speedup monotonically increases with additional threads for all object sizes while the performance of ARIES declines for 8 or more threads.

5.3 Persistent data structure performance

We also evaluate the impact of EAWs on several lightweight 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. 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

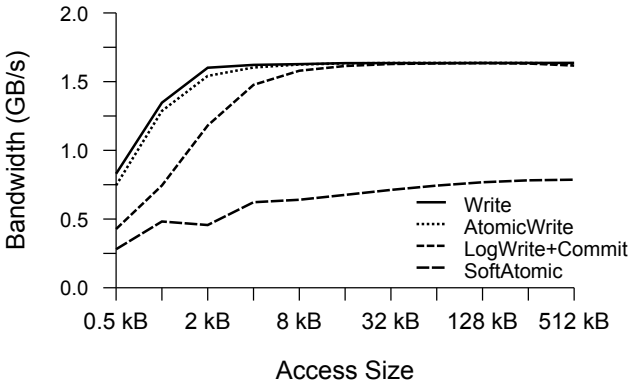


Figure 4: Transaction throughput. By moving the log processing into the storage device, our system is able to achieve transaction throughput nearly equal to normal, non-atomic write throughput.

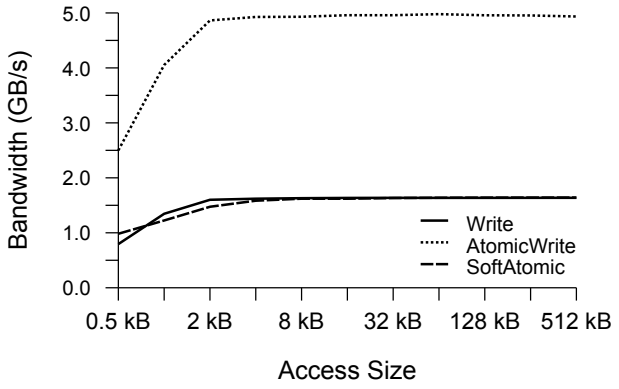


Figure 5: Internal bandwidth. Hardware support for atomic writes allows our system to exploit the internal bandwidth of the storage array for logging and devote the PCIe link bandwidth to transferring useful data.

on node safety and lock coupling. The B+tree is a good case study for EAWs 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 a key-value pair at random.

Six Degrees operates on a large, scale-free graph representing a social network. It alternately searches for six-edge paths between two queried nodes 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 7 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, EAWs sacrifice just 13% of the performance of the unsafe versions on average. Six Degrees, on the other hand, sees a 21% performance drop with EAWs because its transactions are longer and modify multiple nodes. Using EAWs also improves scaling slightly. For instance, the EAW version 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.

5.4 MemcacheDB performance

To understand the impact of EAWs at the application level, we integrated our hash table into MemcacheDB [10], a persistent version of Memcached [25], 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. In this experiment, we run Berkeley DB on Moneta under XFS to make a fair comparison with the EAW-based version. MemcacheDB uses a client-server architecture, and we run both the clients and server on a single machine.

Figure 8 compares the performance of MemcacheDB using our EAW-based hash table as the key-value store to versions that use volatile DRAM, a Berkeley DB 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. Besides the performance gap, DRAM scales better than EAWs with thread count. These differences are due to the disparity between memory and PCIe bandwidth and to the lack of synchronous writes for durability in the DRAM version.

Our system is 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 MemcacheDB uses a single lock for updates.

6 Related Work

Atomicity and durability are critical to storage system design, and 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. We describe existing systems in this area and highlight the differences between them and the system we describe in this work.

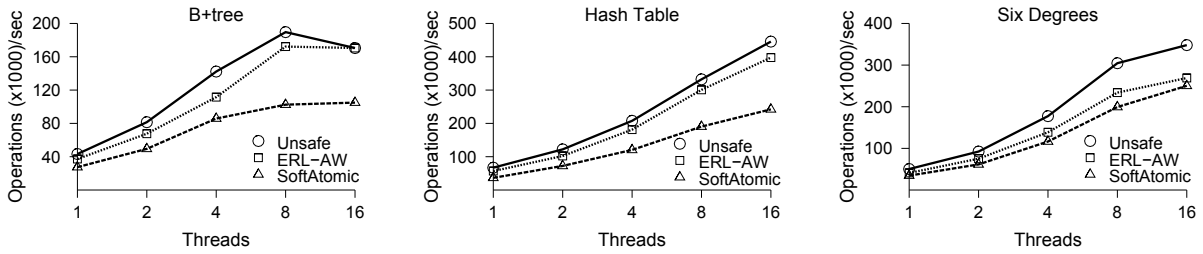


Figure 7: Workload performance. Each set of lines compares the throughput of our (a) B+tree, (b) hash table, and (c) Six Degrees workloads for Unsafe, EAW, and SoftAtomic versions as we scale the number of threads.

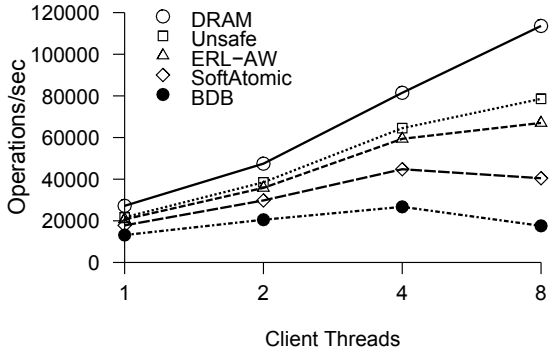


Figure 8: MemcacheDB performance. Adding hardware support for atomicity increases performance by 1.7× for eight clients, and comes within 15% of matching the performance of an unsafe version that provides no durability.

6.1 Disk-based systems

Most disk-oriented systems provide atomicity and durability via software with minimal hardware support. ARIES [26] uses WAL to provide these guarantees and other features while optimizing for the sequential performance of disk. It is ubiquitous in storage and database systems today.

Recent work on segment-based recovery [37] revisits the design of WAL for ARIES with the goal of providing efficient support for application-level objects. By removing LSNs on pages, segment-based recovery enables 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 LSNs and without modifying the format or layout of objects.

Stasis [36] 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 EAWs, but achieving good performance would require significant change to its internal organization.

EAWs provide atomicity and durability at the device level where hardware can enforce them efficiently. The Logical Disk [14] provides a similar interface with atomic recovery units (ARUs) [17], which are an abstraction for failure atomicity for multiple disk writes. Like our system, this abstraction does not guarantee that transactions can execute concurrently and produce correct results. The application must implement concurrency control (e.g., two-phase locking). Unlike our system, ARUs do not provide durability, but they do provide isolation.

File systems including WAFL [18] and ZFS [28] use shadow paging to perform atomic updates. Recent work on a byte-addressable persistent file system (BPFS) [13] extends shadow paging to support fine-grained atomic writes to non-volatile main memory. Shadow paging can be advantageous because it requires writing the data only once and then updating a pointer at commit. However, it is not without cost: data must be organized in a tree and updates often require duplicating portions of the tree. In contrast, EAWs require writing the new data twice but the second write is performed by the hardware so the cost is largely hidden. The hardware does in-place updates which makes data management in software much simpler.

Researchers have provided hardware-supported atomicity for disks. Mime [9] 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 otherwise it is very different. To optimize for disks, Mime is designed to avoid synchronous writes by doing copy-on-write updates to a log. It maintains a block map and additional metadata to keep track of the resulting versions of updated data.

6.2 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) [30] 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. The commit protocol relies on a specific feature of flash: It uses per-page metadata to create a linked list of records that form a cycle when commit completes. 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 [29] proposes an atomic-write interface in a commercial flash-based SSD. Their system uses a log-based mapping layer in the drive's FTL, and it requires that all the writes in one transaction be contiguous in the log. This prevents them from supporting multiple, simultaneous transactions.

6.3 Non-volatile main memory

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

Recoverable Virtual Memory (RVM) [34] 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 that is used to quickly revert the contents of buffered pages without rereading them from disk when a transaction aborts. Rio Vista [24] 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, while MARS supports transactions that scale with the capacity of the storage device. MARS could be coupled with demand paging to serve as the underlying mechanism for the transactions over regions of virtual memory that these systems provide.

More recently, Mnemosyne [39] and NV-heaps [12] provide transactional support for building persistent data structures in byte-addressable NVMs attached to the memory bus. Both systems map storage directly into the application's address space, making it accessible by normal load and store instructions. These systems focus on programming support for NVMs while our work provides a lower-level interface for a storage architecture that supports transactions more efficiently.

7 Discussion

EAWs and MARS are a starting point for thinking about how to leverage the performance of fast NVMs while providing the features that applications require. They also raise several questions that lay the foundation for future work in this area.

MARS currently only supports a database that can fit in the storage of a single machine. But MARS, and EAWs more generally, could be useful for distributed databases and persistent data structures that span many machines because these systems must make guarantees about the persistence of local data. EAWs could accelerate transaction commit on individual nodes and two-phase commit on multiple nodes. It would also be possible to extend EAWs to a network-enabled version of Moneta [8].

MARS does not obviate the need for replication for applications that require high reliability and availability. Replicating data across multiple arrays of fast NVMs is challenging because network latency can dominate storage latency. However, because MARS improves the performance of atomic operations on individual machines, it could benefit the state machine replication approach for fault tolerance [21]. A system such as the Chubby lock service [4] stores each shared object as an entry in a database on each replica. It also maintains a log on each replica to track and execute the Paxos algorithm [22]. MARS could be used to implement both the local database and the log.

To maximize performance, we focus on implementing redundancy within the SSD itself in the form of error correcting SEC-DED codes that allow memories to tolerate some errors. If an application requires higher levels of resilience, RAID or other techniques may be necessary and they may necessitate a different implementation of EAWs.

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, called editable atomic writes (EAW), that takes advantage of the parallelism and performance in fast NVM-based storage. We demonstrated MARS and EAWs 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×. Across a range of persistent data structures, EAWs improve operation throughput by an average of 1.4×. When applied to MARS, EAWs yield a 3.7× performance improvement relative to a baseline implementation of ARIES.

Acknowledgements

We would like to thank the anonymous reviewers, our shepherd, Michael Swift, and Geoff Voelker for their valuable feedback. This work is supported by NSF Award 1219125 and by hardware donations from Xilinx.

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

[4] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[5] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, SIGMOD '94, pages 383–394, New York, NY, USA, 1994. ACM.

[6] 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.

[7] 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.

[8] A. M. Caulfield and S. Swanson. Quicksan: a storage area network for fast, distributed, solid state disks. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 464–474, New York, NY, USA, 2013. ACM.

[9] 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.

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

[11] D. Cobb and A. Huffman. Nvm express and the pci express ssd revolution. http://www.nvmexpress.org/wp-content/uploads/2013/04/IDF-2012-NVM-Express-and-the-PCI-Express-SSD-Revolution.pdf.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the system r database manager. *ACM Comput. Surv.*, 13(2):223–242, June 1981.

[17] 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.

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

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

[20] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 24–35, New York, NY, USA, 2009. ACM.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[22] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[23] 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.

[24] 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.

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

[26] 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.

[27] NVMHCI Work Group. NVM Express. http://nvmexpress.org.

[28] Oracle. Solaris ZFS. https://java.net/projects/solaris-

zfs.

[29] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 301–311, Washington, DC, USA, 2011. IEEE Computer Society.

[30] 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.

[31] 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.

[32] 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.

[33] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[34] 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.

[35] 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.

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

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

[38] Silicon Graphics International. XFS: A high-performance journaling filesystem. http://oss.sgi.com/projects/xfs.

[39] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

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

[41] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.