Beyond Block I/O: Rethinking Traditional Storage Primitives *

Xiangyong Ouyang[†‡], David Nellans[†], Robert Wipfel[†], David Flynn[†], Dhabaleswar K. Panda[‡]
[†] *FusionIO and* [‡]*The Ohio State University*

Abstract

Over the last twenty years the interfaces for accessing persistent storage within a computer system have remained essentially unchanged. Simply put, seek, read and write have defined the fundamental operations that can be performed against storage devices. These three interfaces have endured because the devices within storage subsystems have not fundamentally changed since the invention of magnetic disks. Non-volatile (flash) memory (NVM) has recently become a viable enterprise grade storage medium. Initial implementations of NVM storage devices have chosen to export these same disk-based seek/read/write interfaces because they provide compatibility for legacy applications. We propose there is a new class of higher order storage primitives beyond simple block I/O that high performance solid state storage should support.

One such primitive, atomic-write, batches multiple I/O operations into a single logical group that will be persisted as a whole or rolled back upon failure. By moving write-atomicity down the stack into the storage device, it is possible to significantly reduce the amount of work required at the application, filesystem, or operating system layers to guarantee the consistency and integrity of data. In this work we provide a proof of concept implementation of atomic-write on a modern solid state device that leverages the underlying log-based flash translation layer (FTL). We present an example of how database management systems can benefit from atomic-write by modifying the MySQL InnoDB transactional storage engine. Using this new atomic-write primitive we are able to increase system throughput by 33%, improve the 90th percentile transaction response time by 20%, and reduce the volume of data written from MySQL to the storage subsystem by as much as 43% on industry standard benchmarks, while maintaining ACID transaction semantics.

1 Introduction

Storage interfaces have remained largely unchanged for the last twenty years. The abstraction of reading and writing a 512B block to persistent media has served us well but the advent of non-volatile memory (NVM) has produced a flood of new storage products which no longer rely on spinning magnetic media to persist data. The dominant NVM technology in use today, NAND Flash [19], has performance characteristics that are dissimilar to prior storage media. There are many benefits to NVM technologies, such as fast random reads and low static power consumption. However asymmetric read/write latency and low write-durability do not allow a simple linear mapping of a logical block address (LBA) onto a physical block address (PBA) if high throughput and enterprise class data integrity are desired.

Most high capacity solid state storage (SSS) devices implement a logical to physical mapping within the device known as a flash translation layer (FTL) [15]. The design of this FTL has direct implications on the performance and durability of the SSS device and significant effort [10, 11, 17, 20, 21] has gone into optimizing the FTL for performance, power, durability, or a combination of these properties. Optimization of the FTL is often a complex co-design of hardware and software where, at the highest level, the input to the FTL is a logical block address (LBA) and the output is commands to the NAND-flash media on which the data is stored. The LBA read/write interface to the FTL is a simple way to interact with SSS devices. However, these legacy interfaces force solid state storage to behave merely as a very fast block device; ignoring any potential value or optimizations that could be provided by utilizing unique aspects of the flash translation layer's management of the physical device. We believe the time has come for additional I/O interfaces to be defined that can leverage the FTL to provide new and interesting storage semantics for applications.

In this work we propose one such native storage interface, atomic-write, that allows multiple I/O operations to be issued as a single atomic unit with rollback support. We implement atomic-write by leveraging the log based mapping layer within an existing FTL and show that this new interface can provide additional functionality to the application with no performance penalty over traditional read/write interfaces. We target database management systems (DBMS) as a driving application in need of atomic-write and modify MySQL's InnoDB storage en-

*This work was supported in parts by NSF grants CCF-0621484, CCF-0916302, and CCF-0937842.

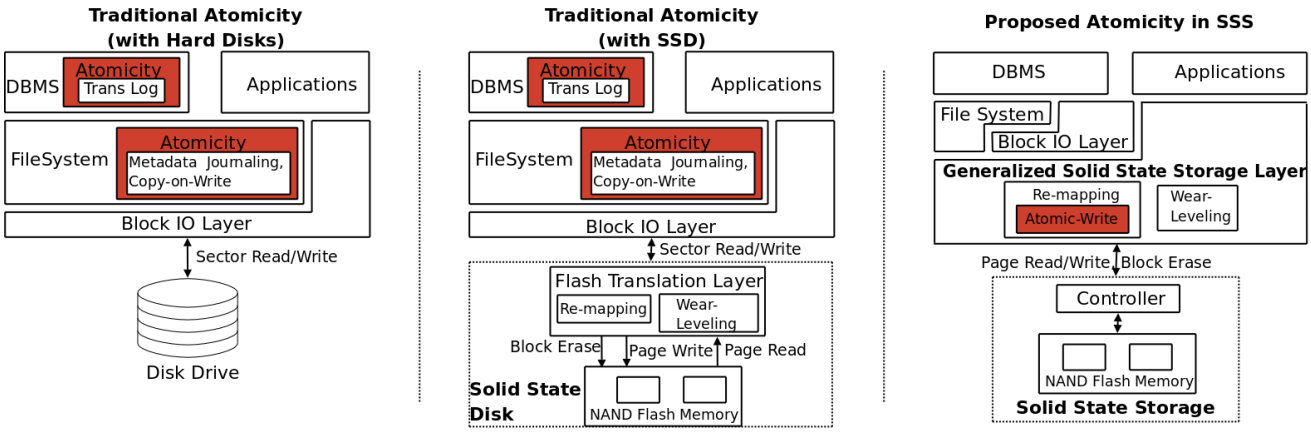

Figure 1. Moving the Atomic-Write Primitive Into The Storage Stack

gine to leverage this new functionality. Using atomic-write we are able to achieve speedups of as much as 33% for the industry standard TPC-C and TPC-H benchmarks. Atomic-write enables a dramatic change in MySQL's I/O patterns required to implement ACID transaction semantics, reducing the need for write-bandwidth by as much as 43%. In addition to improving performance, reducing unnecessary writes has the secondary effect of doubling device longevity due to wearout, eliminating a major barrier to solid state storage adoption in the enterprise market [22].

The examples with MySQL demonstrate the potentials of atomic-write to improve DBMS efficiency while significantly reducing its complexity. Moreover, this new primitive can transparently benefit a wide spectrum of data-intensive applications that require high data consistency and durability. One such example is the hybrid-disk [18], which associates a solid state device to a hard disk to cache the most recently accessed data for fast retrieval. In such an scenario data integrity is required as well as high performance. Atomic-write will be able to unleash the full capacity of the solid state device with data consistency warranted.

Atomic-write is just one of many potential optimizations that can be made by recognizing that log based flash translation layers can provide synergies with many hard to solve problems at the application level. Discard, also known as trim [8, 9], was the first storage interface proposed for applications to communicate higher order I/O intentions to SSS. The storage device could then utilize these application provided hints to optimize allocation and performance. Atomic-write takes a similar approach and exposes a new storage interface in the hope that filesystems, databases, and other applications will leverage this as a building block of transactional systems. By exposing the potential value of one such interface, atomic-write, we hope to encourage the computer architecture community to begin investigating other possible storage interface optimizations beyond simple block I/O.

2 Motivation

The goal of computer architecture is to improve complex systems for real world application benefit. Yet in many cases a system architect is too far removed from the details of application design that information is lost or artificial constraints are imposed that hinder overall improvement. We took the opportunity as storage architects to examine mechanisms that exist to support many classes of applications, with the goal of making a general improvement. One common application method is the *transaction*.

Transactional semantics provide a powerful approach to achieving data integrity as well as concurrency control and crash recovery. The ACID (Atomicity, Consistency, Isolation and Durability) properties [14] which define a strong form of transactional semantics, are highly utilized within many applications, such as filesystems, web services, search engines and scientific computing. Traditionally, supporting transactional semantics has been the primary role of database management systems (DBMS), through the interplay of logs, locks, buffers, and process management. This is well-suited for enterprise applications where the entire system design can be architected from start to finish. However many applications have different storage requirements that do not fit well with the DBMS interface because for efficient access they must control their own data layout and data access mechanisms.

Many studies have been carried out to embed transactional support into portions of the operating system such as the filesystem [13, 27–29, 34] or inside a specialized kernel module [24, 26, 32]. Figure 1 shows the traditional locations of transaction managers within the storage stack. This is a convenience feature that allows applications to take advantage of pre-existing transactional semantics without incurring the complexity and overhead of relying on a full DBMS.

All these transaction managers, regardless of being a DBMS or embedded, are built on the assumption that a

small data unit can be atomically written to persistent storage. This unit of data has historically been a single sector within one block device, a byproduct of the physical device design. As a result, complicated protocols have been designed to chain multiple datum together into useful logical groups, provide data version control across these groups, and allow failure recovery to a known state if the program should crash during the read or writing of these groups.

With the advent of solid state storage, we are now equipped with the capability to redesign this cornerstone of a transactional system. By exploiting the inner workings of the flash translation layer, we are able to extend the conventional write semantics, which only guarantees atomicity for writing one piece of data, to a more general scenario: atomically write multiple non-contiguous pieces of data into persistent storage. Moving the atomic-write primitive out of user space libraries and operating system implementations into the FTL as shown in Figure 1,

In this work, we target the MySQL InnoDB storage engine as a key example of a widely used ACID compliant transaction manager. By modifying InnoDB to utilize the new atomic-write storage primitive, we show any application utilizing MySQL will achieve a significant performance advantage while still maintaining ACID compliance. Re-architecting applications to leverage a storage primitive is substantial work, but in this paper we argue that the gains in performance and decreased data bandwidth will be worth the effort.

The rest of the paper is organized as follows. In Section 3 we describe how atomic-write can be implemented efficiently within a log based FTL. Section 3.3 discusses why atomic-write implemented within the FTL layer is fundamentally more efficient than higher level implementations. We describe the modifications to MySQL that allow it to take advantage of atomic-write in Section 4. Related work and experimental methodology are presented in Sections 5 and 6.1. Experimental results are described in Section 6, showing the efficiency of our atomic-write implementation and how atomic-write affects several industry standard database benchmarks. Finally, we present our conclusions on the value of atomic-write primitives in Section 7.

3 FTL Implementation of Atomic-write

Many of today's advanced solid state storage devices employ a variation of a log structured file system [25] when implementing their flash translation layer (FTL). In log based designs all writes to the media are sequentially appended to the tail of the log and a separate garbage collection thread is responsible for reclaiming deleted/superseded sectors from the head of the log. Log based designs work well for NAND-flash based devices because the slow erase time of physical blocks is no longer on the critical path for write operations. To implement a log based system, the FTL manages a mapping of logical (LBA) to physical block addresses (PBA). Thus, when a logical block is overwritten, the mapping must be updated to point to the new physical block, and the old block must be marked in the log as available for grooming. The garbage collector will eventually erase the block for re-use. Generally, this mechanism works well to provide efficient read and write access to NAND-flash based devices. However it's not able to provide an arbitrarily sized atomic-write guarantee for two reasons:

- A write request may contain data that spans multiple contiguous physical blocks within the NAND-flash. Each physical block within NAND-flash must be programmed as a separate unit, thus requiring iterative or parallel programming, which isn't an atomic operation.
- If multiple sectors are being iteratively written and a system failure occurs, some blocks may be completely written, one block may be partially written, and others will be un-written. The failure recovery process must be able to identify both fully written blocks which should not have been committed, as well as partially written blocks. Incorrect identification of these blocks will result in them being marked as valid within the log, and the superseded data will be erased making future recovery impossible.

In order to overcome these limitations we extend the implementation of a log based FTL to support tracking of committed and uncommitted blocks and the necessary crash recovery semantics that utilize this tracking information.

3.1 Event Log Tracking of Atomic-Writes

Figure 2 provides an example of the tracking methodology we use within the log to identify physical blocks that are part of an atomic-write; we augment the PBA association with a single bit per block to track if any given block is part of an atomic-write operation. Because traditional single block writes are always atomic, this flag is set to "1" for any normal write operation. When the first sector of an atomic-write begins this flag is set to "0", any subsequent physical blocks written as part of the atomic-write operation are also marked as "0", until the final block is handled which will again be marked with the flag set to "1". As a result, the bit tracking fields in the log for an atomic-write form a sequence that's very easy to identify. For example, if an atomic-write consists of 3 sectors, then the flag sequence is "001", as shown in Figure 2.

For this implementation it is a requirement that all blocks belonging to an atomic-write are in contiguous locations within the event log. As a result, data blocks from other write requests are not allowed to interleave with atomic-writes. The benefit of this requirement is that any

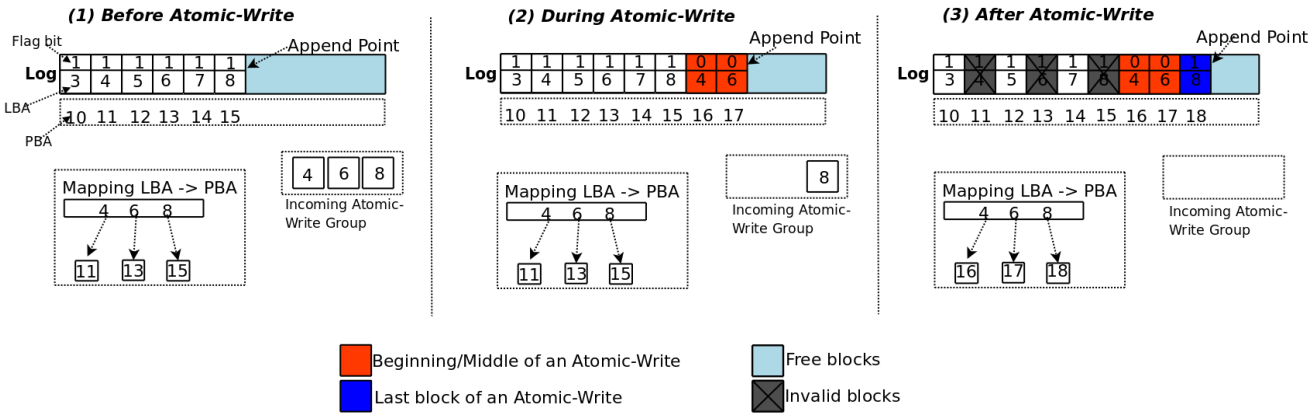

Figure 2. Implementing Atomic Write Within a Log Based FTL

atomic transaction can be easily identified as incomplete if it is not ended by a physical block tagged as "1". We recognize that serializing atomic-writes in the data stream is undesirable, however in practice we have found that there is very little downside to this design choice since applications using atomic-write semantics recognize that large transactions are costly, and thus try to minimize transaction size. Armed with a method of identifying atomic-writes, we must still guarantee that superseded data is not garbage collected from the log before blocks within an atomic-write have been fully committed, and that upon crash recovery the uncommitted atomic-write blocks are removed from the log.

3.2 Delayed Garbage Collection and Crash Recovery

Simply modifying the tracking within the log is not enough to allow rollback to the previous version of data should a write-failure occur. As illustrated in Figure 2, the LBA to PBA mapping table must also be aware of atomic-write semantics since this mapping defines what data is valid, discarded, and superseded, making it available for garbage collection. To prevent valid data from being garbage collected before an atomic-write is fully committed, we simply delay updating this range encoded mapping table until the physical data has been committed to the log. By delaying the mapping table update, previous versions of data will never be erased by the garbage collector until a fully committed atomic-write group is on physical media. In the event of a crash recovery in which the physical blocks were written to disk but the mapping table was not updated, the mapping table can be completely recreated from the log.

During crash-recovery, the log is examined starting at its tail. If the first block contains a "1" then we can safely conclude the storage device was not left in an inconsistent state. If a failure happens in the middle of an atomic-write, we know the log will contain several blocks marked "0" with no "1" preceding it (on the tail). Thus, if the last block written to the log has a "0" flag, we have encountered an incomplete atomic-write. We must then scan backwards, finding all blocks with flag "0" until we encounter the first block with the flag set to "1" which marks the last previous successful completion of either a normal write, or previous atomic-write. All blocks marked with "0" flag must be discarded from the log. Once the tail of the log has been cleaned of any failed atomic-write, a full scan of the log beginning at the head allows us to rebuild the most recent valid data.

Combining the log bitmask, delayed mapping table update/invalidate, and log tail examination upon crash recovery allows us to fully implement atomic-write semantics within our log based FTL.

3.3 Placement Within The Storage Stack

The concepts we use to implement atomic-write in Sections 3.1 and 3.2 have been explored in many other contexts [25, 27, 34]. There are also alternative ways one might implement atomic-write within the storage stack. For instance, ZFS [7] provides a strong guarantee that a write to the filesystem is always atomic by using a copy-on-write model. Other filesystems, such as ext2/3/4 allow files to be opened in append-only mode. Append-only allows the filesystem to guarantee that data in the file will never be overwritten or truncated. These files also grow indefinitely, requiring application control to open the file in a different mode (RW) to eliminate old copies of data which will no longer be referenced. An application could then implement its own tracking of data, much like our log based implementation, to track the most recent copy of data structures written within the file.

The common thread among these high level implementations of atomic-write is that they fundamentally rely on creating multiple copies of on-disk storage, so that previous versions are not over-written. We identify the key insight in this work: *Log based Flash Translation Layers already maintain multiple copies of data within the storage device, thus there is no need to duplicate this effort to implement*

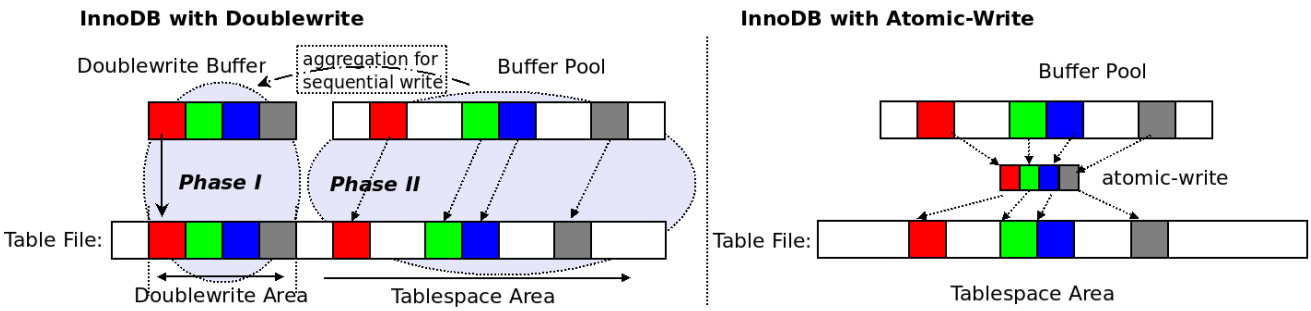

Figure 3. MySQL Disk Accesses to Guarantee Data Integrity

atomic-write at higher levels in the storage stack. As we will show in Section 6, by moving atomicity semantics into the log based FTL the amount of data being written to disk decreases substantially and as a result can substantially improve performance for applications relying on transactional semantics.

4 Database Optimization with Atomic-Write

As discussed in Section 2, database management systems are one class of applications that typically require strong I/O atomicity guarantees. The atomic guarantee on high level logical pages are implemented by systematic control of logs, buffers, and locks on the underlying storage. In this section, we demonstrate how the popular InnoDB [2] database engine for MySQL [4] can leverage the atomic-write primitive to achieve a performance improvement and simplified implementation, without modifying its ACID compliance.

4.1 InnoDB Transaction Logging

Most database implementations use the notion of a transaction log to track changes made to a data page, and InnoDB is no different. InnoDB also utilizes an optimization known as physiological logging [14] to reduce the cost of writing a new event to the transaction log. Physiological logging records only the *deltas* to a data page in its log record, not the full data page, with the goal of minimizing transaction log writes which are inherently a synchronous operation on the critical path for database writes and updates. Utilizing physiological writes enables high throughput to the transaction log. However, because the transactional log doesn't contain a full copy of the data, InnoDB must make a complete copy of the data page before applying the delta. This is required so that there is always a fully valid copy of the previous data page on disk to be able to recover from should a failure occur during the write/update.

The transactional log cannot grow indefinitely and wraps around frequently due to space constraints. Before the tail of the transactional log can be overwritten, the data pages in the tablespace corresponding to those log records must be brought up to date, otherwise the changes performed by the operations represented in those log records will be lost. This is done by applying the dirty page deltas to the tablespace file. If a system failure were to happen during a delta application that leaves only a partially-written page in the tablespace, InnoDB would be unable to recover the original data page resulting in lost data.

4.2 InnoDB Double-write

To overcome the partial-write consistency issues when updating the tablespace, InnoDB utilizes a two phase page update technique known as *double-write*. Figure 3 illustrates the two phases required by double-write to guarantee page consistency.

- In Phase I, InnoDB copies discrete dirty pages from its in memory buffer pool into an additional dedicated in memory buffer area called *double-write buffer*. This contiguous group of memory pages is then written sequentially and synchronously to a dedicated area within the tablespace file, called the double-write area. If write buffering is being used, a fsync, or flush, is called to force the data through all buffering onto persistent media.

- In Phase II, InnoDB re-writes these same individual dirty data pages to their final locations in the tablespace using synchronous random writes since these pages can be scattered throughout the the table space file. If write buffering is being used a fsync is again called to force the data through all buffering onto persistent media.

With this two-phase double-write strategy, InnoDB can guarantee a complete base data page (to which the transaction deltas can be applied) always exists in persistent storage even in the face of a system failure. Should a failure happen that leaves any tablespace data in an inconsistent state, InnoDB will check double-write area, the tablespace area and the transaction log. If a page in the double-write area (Phase I) is found to be partially written, it's simply discarded since the most recent correct copy still exists in the tablespace. If a page in tablespace is inconsistent, which implies a failure in Phase II, it is recovered using the copy of page in double-write area. Once recovered, any

transaction log event can be replayed starting in the appropriate Phase (I or II) that had last completed successfully.

4.3 Double-write Implications on Storage

Double-write is an effective solution to solving the partial-page write issue but it has significant implications on solid state storage.

- Firstly, double-write imposes an additional write phase (Phase I) that is serialized with the in-place update of tablespace data in Phase II. When working with conventional mechanical disks, Phase I, dominated by sequential-write, is much more efficient compared to random-writes in Phase II. Thus the 100% write overhead only results in a small performance degradation. However, advanced solid state storage can achieve random-write performance very close to the performance of sequential-write, shown in Tables 3 and 2. Therefore the overhead of this additional write phase is now much more costly in the era of solid state storage.

- Secondly, double-write is named appropriately because it literally writes every data page twice to stable storage. One of the major functions of a FTL layer is to allow re-mapping of LBA to PBA addresses so that wear-leveling can occur transparently to the application. By performing two writes for every singular data page that is intended to persist on media in one location, the double-write approach effectively halves the useful life of a NAND-flash device which is subject to wear-out effects.

4.4 Replacing Double-write with Atomic-write

InnoDB relies on double-write to protect itself from partial-write of a page (which is made up of multiple physical device blocks). We propose that InnoDB can be modified to replace its complex double-write strategy with the atomic-write primitive described in Section 3.

Figure 3 shows the natural fit of an atomic-write primitive into MySQL. Rather than performing Phase I of the double-write procedure, pages within the tablespace can be overwritten directly using the atomic-write primitive which guarantees that, this compound update will succeed or fails in entirety. If this atomic-write commits, the transaction delta can simply be removed from the transaction log. If it fails, no explicit recovery is required by InnoDB because the storage subsystem will recover the original pages in place and it will appear to InnoDB that no write to the tablespace ever occurred. By implementing atomic-write within the storage subsystem, we *remove the possibility that partial page writes can occur*.

While the InnoDB recovery process is greatly simplified, there is a substantial performance benefit as well. Atomic-write has replaced a series of serialized sequential and random writes, with a single operation containing half the data payload compared to the original implementation. This reduces the backing store bandwidth required by MySQL by half and doubles the effective wear-out life of the solid state storage device.

It's worth noting that atomic-write achieves the aforementioned improvements, namely faster write-completion in critical path and reduced write-wearing of the solid state devices, with a strong guarantee for data integrity. This desirable feature can transparently benefit a large number of data-intensive applications that have data integrity requirements.

5 Related Work

Flash translation layers have received significant study because the LBA to PBA mapping layer is on the critical path for both read and write operations. There have been several efforts to compare the efficiency of block mapping versus page mapping FTL designs [10, 11, 17, 20]. Lim et al. [21] specifically try to improve the performance of a block mapping system to that of a page mapping scheme without requisite memory overhead. Shim et al. [31] attempt to partition the on-board DRAM cache between mapping and data buffering to optimize performance. Seppanen et al. [30] focus on how to optimize the operating system to maximize performance of solid state devices. Our work differs from these in that we are providing a new primitive and additional functionality, not just optimizing performance within the existing paradigm.

Choi et al. [12] and Josephson et al. [16] have both investigated how filesystems might be able to integrate more closely with log based flash translation layers. While closest to our work, both of these require integrating with functionality within the FTL that is not exported for general use. The atomic-write primitive proposed in this work could be leveraged by both studies to help decouple themselves from the FTL. Filesystems such as ZFS [7] and EXT3cow [23] implement a copy-on-write technique to preserve data atomicity which is functionally similar to InnoDB's double-write methodology.

Seltzer et al. [27–29] describe how to support atomicity and transactions within a log structured filesystem. Vijayan et al. [33] studied how to export a transactional interface via a cyclic commit protocol to ensure transactional semantics on top of exisiting SSD devices. All these studies assume that the basic atomic primitive provided by the lowest level of storage is a single fixed 512B or 4KB block. We differ from these works by showing that it is fundamentally more efficient to support multiple block atomicity within the FTL than build atomicity guarantees at higher levels within the storage stack.

6 Experimental Results

6.1 Methodology

The baseline for all results in this paper utilizes an unmodified FusionIO 320GB MLC NAND-flash based device and the most recent production driver available. For this work we implement atomic-write within a research branch of the recently shipped version 2.1 of the FusionIO driver/firmware [1]. We have extended the InnoDB storage engine for MySQL to leverage atomic-write support as described in section 4. All tests are performed on a real machine for which the specification is shown in Table 1, none of our results are simulated.

In Section 6.2 to measure the bandwidth and latency achieved by our atomic-write primitive, we implement a hand tuned microbenchmark designed to minimize control path overhead and unnecessary memory copies. These benchmarks expose the implementation efficiency of atomic-write compared to other I/O methods currently available. After showing atomic-write to be the highest performing I/O method available, in Section 6.3 we evaluate the performance benefits of atomic-write on real database applications, using MySQL 5.1.49 with the InnoDB storage engine. A detailed description of each test can be found in-line with the results.

Table 1. Experimental Machine Configuration

Processor	Xeon X3210 @ 2.13GHz
DRAM	8GB DDR2 677MHz 4x2GB DIMMs
Boot Device	250GB SATA-II 3.0Gb/s
DB Storage Device	FusionIO ioDrive 320GB PCIe 1.0 4x lanes
Operating System	Ubuntu 9.10 - Linux Kernel 2.6.33

6.2 I/O Microbenchmarks

6.2.1 Write Latency

Latency is the round trip time required for an I/O operation to be durably recorded to storage. Minimizing latency is important because many applications perform serialized storage operations on small datum to guarantee transactional consistency (such as MySQL).

To evaluate the control overhead required by various I/O methods available in Linux, we test the total time required to perform a compound write which consists of 64x512B blocks to storage (averaged over 100 iterations). For atomic-write (A-Write) we encapsulate all blocks in a single atomic-write request, issue the request to FTL, then wait for its completion. Since atomic-write does not buffer data there is no need to perform a post write buffer flush.

For synchronous I/O, we serialize the block writes by issuing a fsync following each write. For asynchronous I/O, we utilize the Linux native asynchronous I/O library, *libaio*, to submit all blocks via one I/O request, wait for the operation to complete, and then do a fsync() to flush data to the media if buffering was enabled. Latency is measured from the beginning of the first I/O issued until the completion of all writes, including the fsync if used.

Three different write patterns are tested:

- *Random* - Blocks are randomly scattered within a 1 GB range and aligned to 512B boundaries.
- *Strided* - Blocks start at position N and are separated by fixed 64KB increments.
- *Sequential* - Blocks are positioned sequentially from position N.

Table 2. Write Latency in Microseconds

		I/O Type		
Pattern	**Buffering**	Sync.	Async.	A-Write
Random	Buffered	4,042	1,112	
	directIO	3,542	851	671
Strided	Buffered	4,006	1,146	
	directIO	3,447	857	669
Sequential	Buffered	3,955	330	
	directIO	3,402	898	685

Table 2 shows the average latency to complete these writes using different I/O mechanisms. Asynchronous directIO is the fastest traditional storage method because it avoids an unnecessary copy from user space into the operating system page cache before flushing the data back to disk upon fsync. Atomic-write is able to slightly outperform directIO because it natively takes multiple write ranges and avoid the overhead of performing multiple system calls. Sequential buffered+async I/O appears to be an outlier to the general trends - this is because *libaio* is capable of detecting and merging multiple contiguous IOPs when using buffered I/O, consolidating the I/O into a single, more efficient, IOP at lower levels in the storage stack. Such an optimization is not possible when using directIO. Atomic-write could make a similar optimization but is beyond the scope of this work.

6.2.2 Write Bandwidth

Bandwidth is the maximum sustained data rate that a storage device can achieve by pipelining commands and maximizing the amount of data transferred per control operation. Maximizing bandwidth can be important for applications which have enough latency tolerance to buffer data intended for storage at the application level before writing it to disk. To test achievable bandwidth we utilize the same methodology as in section 6.2.1, however we increase the individual block size from 512B to 16KB to maximize the ratio of data transferred per control word and do not require fsyncs after each write, only a single fsync at the completion of buffered I/O.

In Table 3, much like Table 2, we find that async directIO is able to achieve the highest throughput for traditional storage methods. Again, this is due to being able to pipeline I/O operations and not performing an extraneous copy into the page cache before flushing the data to media. Atomic-write appears to slightly outperform directIO in all cases, but the difference is simply in implementation, atomic-write should have no fundamental performance advantage over asynchronous directIO. It is worth noting that MySQL does not natively use async directIO. Instead, it chooses to use synchronous directIO but implement its own I/O offload thread which approximates the behavior of asynchronous I/O while providing better control over I/O re-ordering for ACID compliance.

Table 3. Write Bandwidth in MB/s

		I/O Type		
Pattern	**Buffering**	Synch.	Async.	A-Write
Random	Buffered	302	301	
	directIO	212	505	513
Strided	Buffered	306	300	
	directIO	217	503	513
Sequential	Buffered	308	304	
	directIO	213	507	514

6.3 Database Workloads

The microbenchmark results in Section 6.2 show that atomic-write can be implemented within a log based FTL and provide performance that meets or exceeds that of the legacy I/O interfaces. To test the real world impact that atomic-write can have on application performance, we evaluate two industry standard transaction processing workloads DBT-2 [5] and DBT-3 [3] which are fair use implementations of TPC-C and TPC-H respectively, and *SysBench* [6], which is another transaction-processing benchmark. The performance metrics we evaluate are: *Transaction Throughput* which is the number of transactions completed per unit time; *Data Written* which is amount of data written to storage during workload execution; and *Latency* which is the average time required for a single transaction to complete.

For this work we configure MySQL to run on a non-shared machine seen in Table 1. InnoDB's buffer pool is set at 4GB and both its transactional log and tablespace files co-exist on the FusionIO device. MySQL's binary log is stored to a separate hard disk drive and is not on the critical path for performance. DBT-2 is configured to use 500 warehouses with a resulting database size of 47GB including indices. DBT-3 is configured with scale factor of 3 resulting in a total database size of 8.5 GB with indices. SysBench uses a table of size 11GB with 20 million tuples in it. The driver for each workload was run on a separate machine connected by 1Gbps Ethernet to avoid polluting the database host system. Each benchmark was run for a minimum of 10 minutes and warmed up once before collecting statistics.

Performance is measured for three distinct test cases:

- MySQL - The unmodified InnoDB engine with double-write enabled. This mode provides full ACID compliance, but shows the performance penalty incurred on a SSS device by having to perform twice the number of writes. This mode is used as the baseline in all results.
- Double-Write Disabled - InnoDB with Phase I of the double-write simply disabled. This is an unsafe mode that may suffer from the "partial-write" problem, but highlights the potential gains of eliminating Phase I of InnoDB's ACID compliant implementation
- Atomic-Write - InnoDB optimized to use *atomic-write* as described in Section 4. Using atomic-write provides the same level of ACID compliance as the baseline InnoDB implementation.

6.3.1 Transaction Throughput

Figure 4 shows the transaction throughput of our three test cases normalized to the baseline InnoDB implementation. Simply disabling phase I of the InnoDB double-write implementation results in a maximum throughput improvement of 9%. Atomic-write is able to outperform the baseline InnoDB implementation by as much as 23%. Both Atomic-write and double-write-disabled write the same amount of data to storage, but InnoDB offloads tablespace writing to an I/O thread which in turn performs synchronous writes. As seen in Table 3, our atomic-write implementation is able to sustain 142% more bandwidth than the native synchronous directIO methodology used by MySQL. As a result, utilizing atomic-write can further improve throughput over simply disabling double-write.

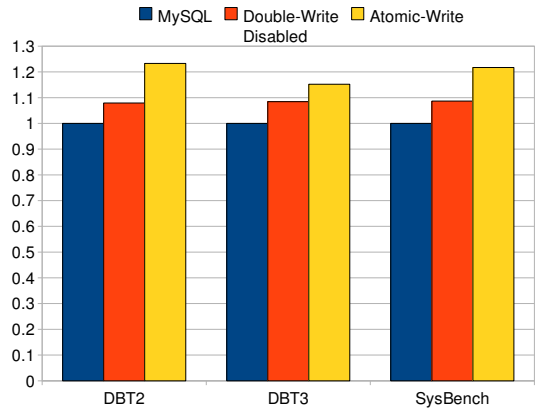

Figure 4. Normalized Throughput

The throughput improvement achievable by using atomic-write within InnoDB is fundamentally limited by

the amount of time spent waiting on write I/O within the workload. There are two factors that affect this wait time: the percentage of read vs. write operations that the workload natively requests - and the amount of memory in use by the database. We defer the sensitivity study of varying the memory to database size ratio and read to write ratio to Section 6.4.

6.3.2 Amount of Data Written to Storage

InnoDB writes both its transaction log and tablespace data to stable storage. Using atomic-write, we are able to optimize the tablespace data storage process reducing the total writes by one half, but the amount of data written to the transaction log is unaffected by either disabling double-write or leveraging atomic-write. Figure 5 shows the relative amount of data written to the underlying storage during workload execution. Disabling double-write from MySQL reduces the total data written to the backing store by up to 46%, while atomic-write reduces total data written by up to 43%. Because atomic-write can process more transactions and generate more write requests during the fixed time execution of the benchmarks, it has slightly higher write-bandwidth. On a per transaction basis, double-write-disabled and atomic-write require the same amount of total I/O.

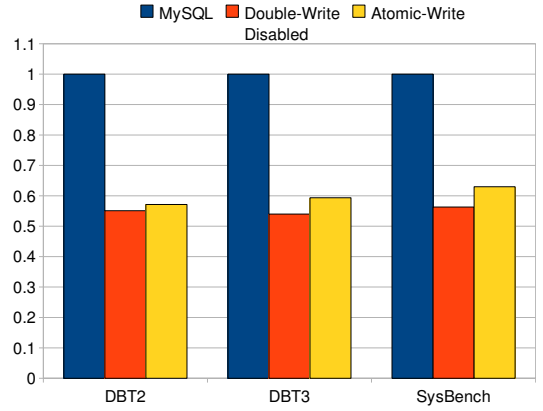

Figure 5. Data Written to Storage (Lower is Better)

In our experimental configuration, each database workload was run in isolation on a single high throughput (>500MB/s) solid state storage device. In enterprise installations the storage subsystem is often shared between one or many applications in a NAS or SAN environment. In these situations, storage subsystem bandwidth is often the single largest bottleneck in database performance; by reducing the write rate to a database by 43%, we also help extend the value of shared storage and network infrastructure. A by-product of reducing the number of writes to storage is that for solid state devices, the useable life of the device is almost doubled. Device wearout has been a major barrier to enterprise adoption, so this significant improvement should not be overlooked.

6.3.3 Transaction Latency

Another important metric in many database driven applications is the average response time per query or transaction. Transaction latency is dominated more by the synchronous write to the transaction log, but write bandwidth also plays an important role. In a memory constraint environment, the database has to frequently flush out dirty data pages to make room for newly accessed pages. This effectively serializes transaction processing with table space writes when they are occurring. Full database checkpointing, which is convenient for crash recovery, effectively blocks all transactions until the tablespace write has finished. Thus, by reducing the amount of data that must be written to storage in both these cases, atomic-write helps decrease both the variation and average latency of transactions. For DBT2 and SysBench, we show the 90th percentile latency in Figure 6. For DBT3 we show the average latency of the queries performed during the execution. Atomic-write is able to reduce 90th percentile latency of DBT2 and SysBench by 20% and 24% respectively. The average latency in DBT3 is reduced by 9%. Many improvements in database throughput often come at the expense of transaction latency. For many interactive database workloads, such as Web 2.0 sites, maintaining a worst case latency is extremely important for usability of the system. Improving both throughput and transaction latency makes atomic-write an ideal database optimization for these types of systems.

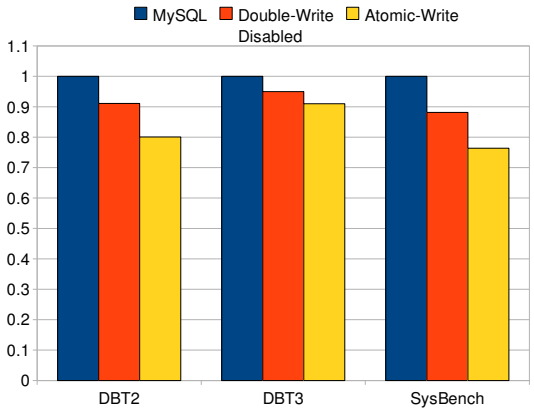

Figure 6. Transaction Latency (Lower is Better)

6.4 Sensitivity Studies

6.4.1 Memory to Database Size

The atomic-write implementation saves 50% of all writes to the tablespace. This does not vary significantly across

the ratio of memory:database sizes seen in Figure 7, because all dirty data must be eventually written back to durable storage. At large ratios of memory buffer to absolute database size, the performance of writing to the tablespace is not on the critical path for database operations. As a result, the application throughput advantage seen when using the atomic-write optimization will vary with the ratio of memory buffer to database size. While in-memory (100% of the database fitting in memory) databases provide optimal performance they are extremely cost ineffective, and most database administrators try to minimize the amount of resources required to achieve acceptable performance. By improving the performance of the I/O subsystem, atomic-write is able to improve performance at any given ratio of memory:database size. Figure 7 compares its performance against the baseline InnoDB with double-write enabled across a wide range of memory:database sizes. The performance gains range from 7% with abundant memory(1:1) to 33% with scarce memory(1:1000) while reducing the volume of data written to disk in all cases by approximately 40%.

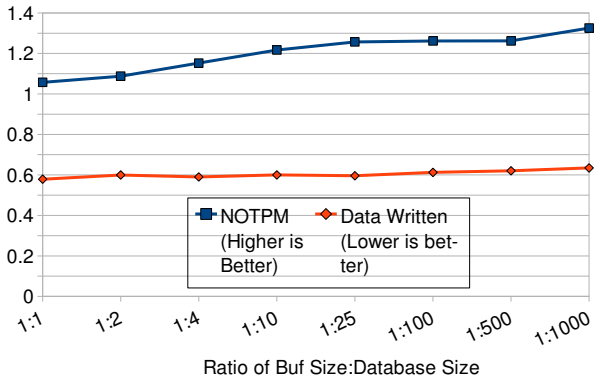

Figure 7. DBT2: Normalized Atomic-Write Performance at Varied Buffer:Database Size

6.4.2 Write/Update Percentage

Another factor that changes the absolute performance achievable with atomic-write is the percentage of read versus write operations a workload performs. A read-only workload produces no dirty data to be flushed to the tablespace, hence it's insensitive to the atomic-write optimization within InnoDB. A write-intensive workload however flushes large volumes of data to storage and is thus heavily dependent on the storage subsystem write-bandwidth. We examine this effect by varying the ratio of update to read operations within each transaction in the SysBench workload. Figure 8 shows both the amount of data written to disk and throughput improvement of atomic-write over the baseline InnoDB implementation when varying the amount of updates in the transaction from 0-100%. With read-only workload(0% updates), only transactional log is written which is unaffected by atomic-write, hence both atomic-write and double-write perform identically. As the workload becomes more write-biased, atomic-write can achieve better improvement in transaction throughput, up to 33%. When ratio of updates to reads per transaction increases, atomic-write enables MySQL to process more transactions per unit time. As a result, more data is physically written to storage including both transaction log and tablespace data, as seen in Figure 8, but per query the 50% savings over double-write to tablespace data remains constant.

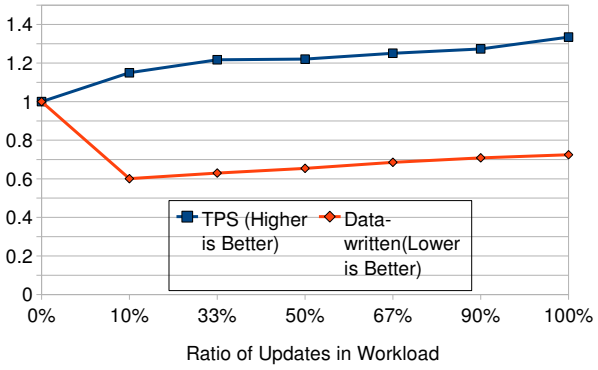

Figure 8. SysBench: Normalized Atomic-Write Performance at Varied Updates:Reads

7 Conclusions and Future Work

In modern computer designs, the storage system is accessed via two verbs - *read* and *write*. For the past twenty years these interfaces have been sufficient because the underlying storage technology has remained largely unchanged. With the introduction of non-volatile memory, the performance and reliability characteristics of storage is changing dramatically. A new class of high performance solid state storage has emerged that implicitly utilize a logical to physical block mapping layer within the device. We propose that this flash translation layer be explicitly recognized so that storage abstractions beyond read and write can be implemented at the most efficient tier possible in the storage hierarchy.

In this work we have implemented one new storage verb, *atomic-write*, within a log-structured FTL that allows multiple I/O operations to commit or rollback as a group. We have shown that the FTL is a natural placement for atomic-write semantics because they can utilize the already existing FTL block tracking mechanisms for commit, rollback, and recovery. We identify ACID compliant transactions as a common application paradigm that can benefit from a high efficiency atomic-write implementation. We demonstrate how the MySQL InnoDB storage

engine can be modified to take advantage of the atomic-write primitive while maintaining ACID compliance. Our InnoDB optimizations results in a 43% reduction in data written to storage, 20% reduction in transaction latency, and a 33% throughput improvement on industry standard database benchmarks.

In future work, we intend to examine how atomic-write can be extended to support multiple outstanding write groups, implementing full transactional semantics. We hope to identify other storage primitives beyond block I/O that have synergies within the FTL and can reduce application complexity, improve performance, and increase storage reliability.

References

[1] FusionIO ioMemory VSL Driver. `http://www.fusionio.com/load/media-docsPress/tm1mj/press_release_optimus_prime.pdf`.

[2] InnoDB Storage Engine. `http://innodb.com`.

[3] MySQL Branch of DBT3. `https://launchpad.net/dbt`.

[4] MySQL Database Server. `http://dev.mysql.com/`.

[5] OSDL: Database Test Suite. `http://osdldbt.sourceforge.net/`.

[6] SysBench. `http://sysbench.sourceforge.net`.

[7] ZFS. `http://www.sun.com/software/solaris/ds/zfs.jsp`.

[8] FusionIO - U.S. Patent Application Pub. No. 2008/0140909. 2008.

[9] FusionIO - U.S. Patent Application Pub. No. 2008/0140910. 2008.

[10] S. N. A Kawaguchi and H. Motoda. A Space-efficient Flash Translation Layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics*, May 2002.

[11] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A Design for High-Performance Flash Disks. *Operating Systems Review*, April 2007.

[12] H. J. Choi, S. ho Lim, and K. H. Park. JFTL: a Flash Translation Layer Based on a Journal Remapping for Flash Memory. *In ACM Transactions on Storage*, 2009.

[13] E. Gal and S. Toledo. A Transactional Flash File System for Microcontrollers. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 7–7, 2005.

[14] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[15] Intel Corporation. Understanding the Flash Translation Layer (FTL) specification. In *`http://developer.intel.com/`*.

[16] W. Josephson, L. Bongo, D. Flynn, and K. Li. DFS: A File System for Virtualized Flash Storage. *In Proceedings of USENIX: FAST*, 2010.

[17] J. Kang, H. Jo, J. Kim, and J. Lee. A Superblock-based Flash Translation Layer for NAND Flash Memory. *In Proceedings of EMSOFT*, 2006.

[18] Kgil, Taeho and Roberts, David and Mudge, Trevor. Improving NAND Flash Based Disk Caches. In *Proceedings of ISCA '08*, 2008.

[19] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A Space Efficient Flash Translation Layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics*, 48:366–375, 2002.

[20] S. Lee, D. Park, T. Chung, S. Park, and H. Song. A Log Buffer-based Flash Translation Layer using Fully-associative Sector Translation. *ACM Transactions on Embedded computer Systems*, 6(3), 2007.

[21] S. Lim, S. Lee, and B. Moon. FASTER FTL for Enterprise-Class flash Memory SSDs. *International Workshop on Storage Network Architecture and Parallel I/Os*, 2010.

[22] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. *4th ACM European Conference on Computer Systems*, 2009.

[23] Z. Peterson and R. Burns. Ext3cow: A Time-Shifting File System for Regulatory Compliance. *ACM Transactions on Storage*, 2, 2005.

[24] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating Systems Transactions. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 161–176, 2009.

[25] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10, 1992.

[26] 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.

[27] M. Seltzer and M. Stonebraker. Transaction Support in Read Optimized and Write Optimized File Systems. In *Proceedings of the sixteenth international conference on Very Large Databases*, pages 174–185, 1990.

[28] M. I. Seltzer. File System Performance and Transaction Support. Technical report, UC Berkeley, 1993.

[29] M. I. Seltzer. Transaction Support in a Log-Structured File System. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 503–510, 1993.

[30] E. Seppanen, M. O'Keef, and D. Lilja. High Performance solid State Storage Under Linux. *Symposium on Massive Storage Systems and Technologies*, 2010.

[31] H. Shim, B. Seo, J. Kim, and S. Maeng. An Adaptive Partitioning Scheme for DRAM-based Cache in Sold State Drives. *Symposium on Massive Storage Systems and Technologies*, 2010.

[32] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright. Enabling Transactional File Access via Lightweight Kernel Extensions. In *Proceedings of USENIX: FAST*, pages 29–42, 2009.

[33] Vijayan Prabhakaran, Thomas L. Rodeheffer and Lidong, Zhou. Transactional Flash. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, 2008.

[34] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID Semantics to the File System. *ACM Transactions on Storage*, 3(2):4, 2007.