Isotope: ACID Transactions for Block Storage

JI-YONG SHIN, Cornell University and Yale University
MAHESH BALAKRISHNAN, Yale University
TUDOR MARIAN, Google, Inc.
HAKIM WEATHERSPOON, Cornell University

Existing storage stacks are top heavy and expect little from block storage. As a result, new high-level storage abstractions—and new designs for existing abstractions—are difficult to realize, requiring developers to implement from scratch complex functionality such as failure atomicity and fine-grained concurrency control. In this article, we argue that pushing transactional isolation into the block store (in addition to atomicity and durability) is both viable and broadly useful, resulting in simpler high-level storage systems that provide strong semantics without sacrificing performance. We present Isotope, a new block store that supports ACID transactions over block reads and writes. Internally, Isotope uses a new multiversion concurrency control protocol that exploits fine-grained, subblock parallelism in workloads and offers both strict serializability and snapshot isolation guarantees. We implemented several high-level storage systems over Isotope, including two key-value stores that implement the LevelDB API over a hash table and B-tree, respectively, and a POSIX file system. We show that Isotope's block-level transactions enable systems that are simple (100s of lines of code), robust (i.e., providing ACID guarantees), and fast (e.g., 415MB/s for random file writes). We also show that these systems can be composed using Isotope, providing applications with transactions across different high-level constructs such as files, directories, and key-value pairs.

Categories and Subject Descriptors: H.2.4 [**Systems**]: Transaction Processing; D.4.2 [**Storage Management**]: Secondary Storage, Storage Hierarchies

General Terms: Design, Experimentation, Management, Performance

Additional Key Words and Phrases: Transaction, block storage, isolation

ACM Reference Format:
Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. 2017. Isotope: ACID transactions for block storage. ACM Trans. Storage 13, 1, Article 4 (February 2017), 25 pages.
DOI: http://dx.doi.org/10.1145/3032967

1. INTRODUCTION

With the advent of multicore machines, storage systems such as file systems, key-value stores, graph stores, and databases are increasingly parallelized over dozens of cores. Such systems run directly over raw block storage but assume very little about its

This work is partially funded and supported by a SLOAN Research Fellowship received by Hakim Weatherspoon, a Facebook Faculty Award received by Mahesh Balakrishnan, DARPA MRC (FA8750-11-2-0256) and CSSG (D11AP00266), NSF (0424422, 1047540, 1053757, 1151268, 1422544), NIST (60NANB15D327), Cisco, and Intel. A conference version of this article appeared in the Proceedings of the USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, February 22-25, 2016.
Authors' addresses: J.-Y. Shin and M. Balakrishnan, Department of Computer Science, Yale University, New Haven, CT 06511; email: {jyshin, mahesh}@cs.yale.edu; T. Marian, Google, Inc., 1600 Amphitheatre Pkwy, Mountain View, CA 94043; email: tudorm@google.com; H. Weatherspoon, Department of Computer Science, Cornell University, Ithaca, NY 14853; email: hweather@cs.cornell.edu.

ACM 1553-3077/2017/02-ART4 $15.00
DOI: http://dx.doi.org/10.1145/3032967

interface and semantics; usually, the only expectations from the block store are durability and single-operation, single-block linearizability. As a result, each system implements complex code to layer high-level semantics such as atomicity and isolation over the simple block address space. While multiple systems have implemented transactional atomicity within the block store [Chao et al. 1992; De Jonge et al. 1993; Prabhakaran et al. 2008; SanDisk 2015a; Coburn et al. 2013], concurrency control has traditionally been delegated to the storage system above the block store.

In this article, we propose the abstraction of a transactional block store that provides isolation in addition to atomicity and durability. A number of factors make isolation a prime candidate for demotion down the stack.

(1) Isolation is *general*; since practically every storage system has to ensure safety under concurrent accesses, an isolation mechanism implemented within the block layer is broadly useful.
(2) Isolation is *hard*, especially for storage systems that need to integrate fine-grained concurrency control with coarse-grained durability and atomicity mechanisms (e.g., see ARIES [Mohan et al. 1992]); accordingly, it is better provided via a single, high-quality implementation within the block layer.
(3) Block-level transactions allow storage systems to effortlessly provide end-user applications with transactions over high-level constructs such as files or key-value pairs.
(4) Block-level transactions are oblivious to software boundaries at higher levels of the stack and can seamlessly span multiple layers, libraries, threads, processes, and interfaces. For example, a single transaction can encapsulate an end application's accesses to an in-process key-value store, an in-kernel file system, and an out-of-process graph store.
(5) Finally, multiversion concurrency control (MVCC) [Bernstein et al. 1987] provides superior performance and liveness in many cases but is particularly hard to implement for storage systems since it requires them to maintain a multiversioned state; in contrast, many block stores (e.g., log-structured designs) are already internally multiversioned.

Block-level isolation is enabled and necessitated by recent trends in storage. Block stores have evolved over time. They are increasingly implemented via a combination of host-side software and device firmware [Microsoft 2016a; Fusion-io 2015]; they incorporate multiple heterogeneous physical devices under a single address space [Soundararajan et al. 2010; Shin et al. 2013]; they leverage new NVRAM technologies to store indirection metadata; and they provide sophisticated functionality such as virtualization [Microsoft 2016a; Stein 2005], tiering [Microsoft 2016a], deduplication, and wear leveling. Unfortunately, storage systems such as file systems continue to assume minimum functionality from the block store, resulting in redundant, complex, and inefficient stacks where layers constantly tussle with each other [Stein 2005]. A second trend that argues for pushing functionality from the file system to a lower layer is the increasing importance of alternative abstractions that can be implemented directly over block storage, such as graphs, key-value pairs [Seagate 2016], tables, caches [Saxena et al. 2012b], tracts [Nightingale et al. 2012], byte-addressable [Badam and Pai 2011] and write-once [Balakrishnan et al. 2012] address spaces, and so forth.

To illustrate the viability and benefits of block-level isolation, we built Isotope, a transactional block store that provides isolation (with a choice of strict serializability or snapshot isolation) in addition to atomicity and durability. Isotope is implemented as an in-kernel software module running over commodity hardware, exposing a conventional block read/write interface augmented with *beginTX/endTX* IOCTLs to demarcate transactions. Transactions execute speculatively and are validated by Isotope on *endTX*

by checking for conflicts. To minimize the possibility of conflict-related aborts, applications can provide information to Isotope about which subparts of each 4KB block are read or written, allowing Isotope to perform conflict detection at subblock granularity.

Internally, Isotope uses an in-memory multiversion index over a persistent log to provide each transaction with a consistent, point-in-time snapshot of a block address space. Reads within a transaction execute against this snapshot, while writes are buffered in RAM by Isotope. When *endTX* is called, Isotope uses a new MVCC commit protocol to determine if the transaction commits or aborts. The commit/abort decision is a function of the timestamp-ordered stream of recently proposed transactions, as opposed to the multiversion index; as a result, the protocol supports arbitrarily fine-grained conflict detection without requiring a corresponding increase in the size of the index. When transactions commit, their buffered writes are flushed to the log, which is implemented on an array of physical drives [Shin et al. 2013], and reflected in the multiversion index. Importantly, aborted transactions do not result in any write I/O to persistent storage.

Storage systems built over Isotope are simple, stateless, shim layers that focus on mapping some variable-sized abstraction—such as files, tables, graphs, and key-value pairs—to a fixed-size block API. We describe several such systems in this article, including a key-value store based on a hash table index, one based on a B-tree, and a POSIX user-space file system. These systems do not have to implement their own fine-grained locking for concurrency control and logging for failure atomicity. They can expose transactions to end applications without requiring any extra code. Storage systems that reside on different partitions of an Isotope volume can be composed with transactions into larger end applications.

Block-level isolation does have its limitations. Storage systems built over Isotope cannot share arbitrary, in-memory soft states such as read caches across transaction boundaries, since it is difficult to update such state atomically based on the outcome of a transaction. Instead, they rely on block-level caching in Isotope by providing hints about which blocks to cache. We found this approach well suited for both the file system application (which cached inode blocks, indirection blocks, and allocation maps) and the key-value stores (which cached their index data structures). In addition, information is invariably lost when functionality is implemented at a lower level of the stack: Isotope cannot leverage properties such as commutativity and idempotence while detecting conflicts.

This article makes the following contributions:

—We revisit the end-to-end argument for storage stacks with respect to transactional isolation, in the context of modern hardware and applications.
—We propose the abstraction of a fully transactional block store that provides isolation, atomicity, and durability. While others have explored block-level transactional atomicity [Chao et al. 1992; De Jonge et al. 1993; Prabhakaran et al. 2008; Coburn et al. 2013], this is the first proposal for block-level transactional isolation.
—We realize this abstraction in a system called Isotope via a new MVCC protocol. We show that Isotope exploits subblock concurrency in workloads to provide a high commit rate for transactions and high I/O throughput.
—We describe storage systems built using Isotope transactions—two key-value stores and a file system—and show that they are simple, fast, and robust, as well as composable via Isotope transactions into larger end applications.

2. MOTIVATION

Block-level isolation is an idea whose time has come. In the 1990s, the authors of Rio Vista (a system that provided atomic transactions over a persistent memory

abstraction) wrote in Lowell and Chen [1997]: "We believe features such as serializability are better handled by higher levels of software. . . . Adopting any concurrency control scheme would penalize the majority of applications, which are single-threaded and do not need locking." Today, applications run on dozens of cores and are multithreaded by default; isolation is a universal need, not a niche feature.

Isolation is simply the latest addition to a long list of features provided by modern block stores: caching, tiering, mapping, virtualization, deduplication, and atomicity. This explosion of features has been triggered partly by the emergence of software-based block layers, ranging from flash FTLs [Fusion-io 2015] to virtualized volume managers [Microsoft 2016a]. In addition, the block-level indirection necessary for many of these features has been made practical and inexpensive by hardware advances in the last decade. In the past, smart block devices such as HP AutoRAID [Wilkes et al. 1996] were restricted to enterprise settings due to their reliance on battery-backed RAM; today, SSDs routinely implement indirection in FTLs, using supercapacitors to flush metadata and data on a power failure. Software block stores in turn can store metadata on these SSDs, on raw flash, or on derivatives such as flash-backed RAM [Jose et al. 2013] and Auto-Commit Memory [SanDisk 2015b].

What about the end-to-end argument? We argue that block-level isolation passes the litmus test imposed by the end-to-end principle [Saltzer et al. 1984] for pushing functionality down the stack: it is broadly useful, is efficiently implementable at a lower layer of the stack with negligible performance overhead, and leverages machinery that already exists at that lower layer. The argument regarding utility is obvious: pushing functionality down the stack is particularly useful when it is general enough to be used by the majority of applications, which is the case for isolation or concurrency control. However, the other motivations for a transactional block store require some justification.

Isolation is hard. Storage systems typically implement pessimistic concurrency control via locks, opening the door to a wide range of aberrant behavior such as deadlocks and livelocks. This problem is exacerbated when developers attempt to extract more parallelism via fine-grained locks, and when these locks interact with coarse-grained failure atomicity and durability mechanisms [Mohan et al. 1992]. Transactions can provide a simpler programming model that supplies isolation, atomicity, and durability via a single abstraction. Additionally, transactions decouple the policy of isolation—as expressed through *beginTX*/*endTX* calls—from the concurrency control mechanism used to implement it under the hood.

Isolation is harder when exposed to end applications. Storage systems often provide concurrency control APIs over their high-level storage abstractions; for example, NTFS offers transactions over files, while Linux provides file-level locking. Unfortunately, these high-level concurrency control primitives often have complex, weakened, and idiosyncratic semantics [Pennarun 2016]; for instance, NTFS provides transactional isolation for accesses to the same file, but not for directory modifications, while a Linux *fcntl* lock on a file is released when any file descriptor for that file is closed by a process [fcn 2016]. The complex semantics are typically a reflection of a complex implementation, which has to operate over high-level constructs such as files and directories. In addition, composability is challenging if each storage system implements isolation independently: for example, it is impossible to do a transaction over an NTFS file and a Berkeley DB key-value pair.

Isolation is even harder when multiversion concurrency control is required. In many cases, pessimistic concurrency control is slow and prone to liveness bugs; for example, when locks are exposed to end applications directly or via a transactional interface, the application could hang while holding a lock. Optimistic concurrency control [Kung

```
/*** Transaction API ***/
int beginTX();
int endTX();
int abortTX();
//POSIX read/write commands
/*** Optional API ***/
//release ongoing transaction and return handle
int releaseTX();
//take over a released transaction
int takeoverTX(int tx_handle);
//mark byte range accessed by last read/write
int mark_accessed(off_t blknum, int start, int size);
//request caching for blocks
int please_cache(off_t blknum);
```

Fig. 1. The Isotope API.

and Robinson 1981] works well in this case, ensuring that other transactions can proceed without waiting for the hung process. Multiversion concurrency control works even better, providing transactions with stable, consistent snapshots (a key property for arbitrary applications that can crash if exposed to inconsistent snapshots [Guerraoui and Kapalka 2008]); allowing read-only transactions to always commit [Bernstein et al. 1987]; and enabling weaker but performant isolation levels such as snapshot isolation [Berenson et al. 1995].

However, switching to multiversion concurrency control can be difficult for storage systems due to its inherent need for multiversion state. High-level storage systems are not always intrinsically multiversioned (with notable exceptions such as WAFL [Hitz et al. 1994] and other copy-on-write file systems), making it difficult for developers to switch from pessimistic locking to a multiversion concurrency control scheme. Multiversioning can be particularly difficult to implement for complex data structures used by storage systems such as B-trees, requiring mechanisms such as tombstones [Driscoll et al. 1986; Reid et al. 2011].

In contrast, multiversioning is relatively easy to implement over the static address space provided by a block store (e.g., no tombstones are required since addresses can never be "deleted"). Additionally, many block stores are already multiversioned in order to obtain write sequentiality: examples include log-structured disk stores, shingled hard drives [Aghayev and Desnoyers 2015], and SSDs.

3. THE ISOTOPE API

The basic Isotope API is shown in Figure 1: applications can use standard POSIX calls to issue reads and writes to 4KB blocks, bookended by *beginTX*/*endTX* calls. The *beginTX* call establishes a snapshot; all reads within the transaction are served from that snapshot. Writes within the transaction are speculative. Each transaction can view its own writes, but the writes are not made visible to other concurrent transactions until the transaction commits. The *endTX* call returns true if the transaction commits, and false otherwise. The *abortTX* allows the application to explicitly abort the transaction. The application can choose one of two isolation semantics on startup: strict serializability or snapshot isolation.

The Isotope API implicitly associates transaction IDs with user-space threads, instead of augmenting each call signature in the API with an explicit transaction ID that the application supplies. We took this route to allow applications to use the existing, highly optimized POSIX calls to read and write data to the block store. The control

```
isofs_inode_num ino;
unsigned char *buf;
//allocate buf, set ino to parameter
...
int blknum = inode_to_block(ino);
txbegin:
beginTX();
if(!read(blknum, buf)){
        abortTX();
        return EIO;
}
mark_accessed(blknum, off, sizeof(inode));
//update attributes
...
if(!write(blknum, buf)){
        abortTX();
        return EIO;
}
mark_accessed(blknum, off, sizeof(inode));
if(!endTX()) goto txbegin;
```

Fig. 2. Example application: *setattr* code for a file system built over Isotope.

API for starting, committing, and aborting transactions is implemented via IOCTLs. To allow transactions to execute across different threads or processes, Isotope provides additional APIs via IOCTLs: *releaseTX* disconnects the association between the current thread and the transaction and returns a temporary transaction handle. A different thread can call *takeoverTX* with this handle to associate itself with the transaction.

Isotope exposes two other optional calls via IOCTLs. After reading or writing a 4KB block within a transaction, applications can call *mark_accessed* to explicitly specify the accessed byte range within the block. This information is key for fine-grained conflict detection; for example, a file system might mark a single inode within an inode block or a single byte within a data allocation bitmap. Note that this information cannot be inferred implicitly by comparing the old and new values of the 4KB block; the application might have overwritten parts of the block without changing any bits. The second optional call is *please_cache*, which lets the application request Isotope to cache specific blocks in RAM; we discuss this call in detail later in the article. Figure 2 shows a snippet of application code that uses the Isotope API (the *setattr* function from a file system).

If a read or write is issued outside a transaction, it is treated as a singleton transaction. In effect, Isotope behaves like a conventional block device if the reads and writes issued to it are all nontransactional. In addition, Isotope can preemptively abort transactions to avoid buggy or malicious applications from hoarding resources within the storage subsystem. When a transaction is preemptively aborted, any reads, writes, or control calls issued within it will return error codes, except for *endTX*, which will return false, and *abortTX*.

Transactions can be nested (i.e., a *beginTX*/*endTX* pair can have other pairs nested within it) with the simple semantics that the internal transactions are ignored. A nested *beginTX* does not establish a new snapshot, and a nested *endTX* always succeeds without changing the persistent state of the system. A nested *abortTX* causes any further activity in the transaction to return error codes until all the enclosing *abortTX*/*endTX* have been called. This behavior is important for allowing storage

systems to expose transactions to end-user applications. In the example of the file system, if an end-user application invokes *beginTX* (either directly on Isotope or through a file-system-provided API) before calling the *setattr* function in Figure 2 multiple times, the internal transactions within each *setattr* call are ignored and the entire ensemble of operations will commit or abort.

3.1. Composability

As stated earlier, a primary benefit of a transactional block store is its obliviousness to the structure of the software stack running above it, which can range from a single-threaded application to a composition of multithreaded application code, library storage systems, out-of-process daemons, and kernel modules. The Isotope API is designed to allow block-level transactions to span arbitrary compositions of different types of software modules. We describe some of these composition patterns in the context of a simple photo storage application called ImgStore, which stores photos and their associated metadata in a key-value store.

In the simplest case, ImgStore can store images and various kinds of metadata as key-value pairs in IsoHT, which in turn is built over an Isotope volume using transactions. Here, a single transaction-oblivious application (ImgStore) runs over a single transaction-aware library-based key-value storage system (IsoHT).

Cross-layer: ImgStore may want to atomically update multiple key-value pairs in IsoHT; for example, when a user is tagged in a photo, ImgStore may want to update a photo-to-user mapping as well as a user-to-photo mapping, stored under two different keys. To do so, ImgStore can encapsulate calls to IsoHT within Isotope *beginTX*/*endTX* calls, leveraging nested transactions.

Cross-thread: In the simplest case, ImgStore executes each transaction within a single thread. However, if ImgStore is built using an event-driven library that requires transactions to execute across different threads, it can use the *releaseTX*/*takeoverTX* calls.

Cross-library: ImgStore may find that IsoHT works well for certain kinds of accesses (e.g., retrieving a specific image), but not for others such as range queries (e.g., finding photos taken between March 4 and May 10, 2015). Accordingly, it may want to spread its state across two different library key-value stores, one based on a hash table (IsoHT) and another on a B-tree (IsoBT), for efficient range queries. When a photo is added to the system, ImgStore can transactionally call *put* operations on both stores. This requires the key-value stores to run over different partitions on the same Isotope volume.

Cross-process: For various reasons, ImgStore may want to run IsoHT in a separate process and access it via an IPC mechanism: for example, to share it with other applications on the same machine or to isolate failures in different code bases. To do so, ImgStore has to call *releaseTX* and pass the returned transaction handle via IPC to IsoHT, which then calls *takeoverTX*. This requires IsoHT to expose a transaction-aware IPC interface for calls that occur within a transactional context.

4. DESIGN AND IMPLEMENTATION

Figure 3 shows the major components of the Isotope design. Isotope internally implements an in-memory multiversion index (*B* in the figure) over a persistent log (*E*). Versioning is provided by a timestamp counter (*A*), which determines the snapshot seen by a transaction as well as its commit timestamp. This commit timestamp is used by a decision algorithm (*D*) to determine if the transaction commits or not. Writes issued within a transaction are buffered (*C*) during its execution and flushed to the log if the transaction commits. We now describe the interaction of these components.

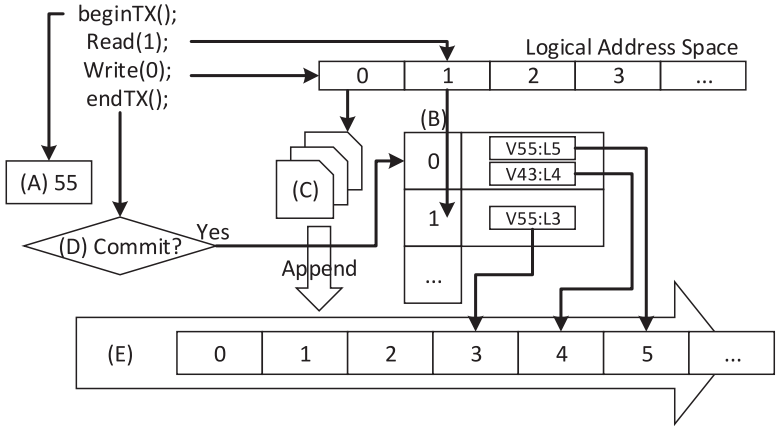

Fig. 3. Isotope consists of (A) a timestamp counter, (B) a multiversion index, (C) a write buffer, (D) a decision algorithm, and (E) a persistent log.

When the application calls *beginTX*, Isotope creates an in-memory intention record for the speculative transaction: a simple data structure with a start timestamp and a read/write-set. Each entry in the read/write-set consists of a block address, a bitmap that tracks the accessed status of smaller fixed-size chunks or fragments within the block (by default, the fragment size is 16 bytes, resulting in a 256-bit bitmap for each 4KB block), and an additional 4KB payload only in the write-set. These bitmaps are never written persistently and are only maintained in-memory for currently executing transactions. After creating the intention record, the *beginTX* call sets its start timestamp to the current value of the timestamp counter (*A* in Figure 3) without incrementing it.

Until *endTX* is called, the transaction executes speculatively against the (potentially stale) snapshot, without any effect on the shared or persistent state of the system. Writes update the write-set and are buffered in-memory (*C* in Figure 3) without issuing any I/O. A transaction can read its own buffered writes, but all other reads within the transaction are served from the snapshot corresponding to the start timestamp using the multiversion index (*B* in Figure 3). The *mark_accessed* call modifies the bitmap for a previously read or written block to indicate which bits the application actually touched. Multiple *mark_accessed* calls have a cumulative effect on the bitmap. At any point, the transaction can be preemptively aborted by Isotope simply by discarding its intention record and buffered writes. Subsequent reads, writes, and *endTX* calls will be unable to find the record and return an error code to the application.

All the action happens on the *endTX* call, which consists of two distinct phases: *deciding* the commit/abort status of the transaction and *applying* the transaction (if it commits) to the state of the logical address space. Regardless of how it performs these two phases, the first action taken by *endTX* is to assign the transaction a commit timestamp by reading and incrementing the global counter. The commit timestamp of the transaction is used to make the commit decision and is also used as the version number for all the writes within the transaction if it commits. We use the terms "timestamp" and "version number" interchangeably in the following text.

4.1. Deciding Transactions

To determine whether the transaction commits or aborts, *endTX* must detect the existence of conflicting transactions. The isolation guarantee provided—strict serializability or snapshot isolation—depends on what constitutes a conflicting transaction. We

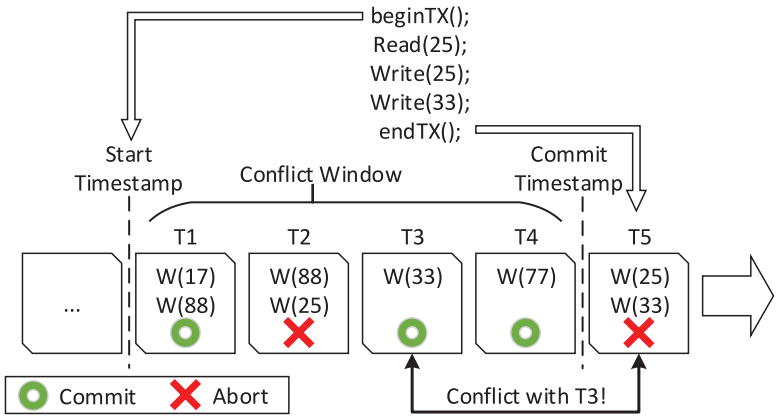

Fig. 4. Conflict detection under snapshot isolation: a transaction commits if no other committed transaction in its conflict window has an overlapping write-set.

first consider a simple strawman scheme that provides strict serializability and implements conflict detection as a function of the multiversion index. Here, transactions are processed in commit timestamp order, and for each transaction the multiversion index is consulted to check if any of the logical blocks in its read-set has a version number greater than the current transaction's start timestamp. In other words, we check whether any of the blocks read by the transaction has been updated since it was read.

This scheme is simple but suffers from a major drawback: the granularity of the multiversion index has to match the granularity of conflict detection. For example, if we want to check for conflicts at 16-byte grain, the index has to track version numbers at 16-byte grain as well; this blows up the size of the in-memory index by 256× compared to a conventional block-granular index. As a result, this scheme is not well suited for fine-grained conflict detection.

To perform fine-grained conflict detection while avoiding this blow-up in the size of the index, Isotope instead implements conflict detection as a function over the temporal stream of prior transactions (see Figure 4). Concretely, each transaction has a conflict window of prior transactions between its start timestamp and its commit timestamp.

—For strict serializability, the transaction T aborts if any committed transaction in its conflict window modified an address that T *read*; otherwise, T commits.
—For snapshot isolation, the transaction T aborts if any committed transaction in its conflict window modified an address that T *wrote*; otherwise, T commits.

In either case, the commit/abort status of a transaction is a function of a window of transactions immediately preceding it in commit timestamp order.

When *endTX* is called on T, a pointer to its intention record is inserted into the slot corresponding to its commit timestamp in an in-memory array. Since the counter assigns contiguous timestamps, this array has no holes; each slot is eventually occupied by a transaction. At this point, we do not yet know the commit/abort status of T and have not issued any write I/O, but we have a start timestamp and a commit timestamp for it. Each slot is guarded by its own lock.

To decide if T commits or aborts, we simply look at its conflict window of transactions in the in-memory array (i.e., the transactions between its start and commit timestamps). We can decide T's status once all these transactions have decided. T commits if each transaction in the window either aborts or has no overlap between its

read/write-set and T's read/write-set (depending on the transactional semantics). Since each read/write-set stores fine-grained information about which fragments of the block are accessed, this scheme provides fine-grained conflict detection without increasing the size of the multiversion index.

Defining the commit/abort decision for a transaction as a function of other transactions is a strategy as old as optimistic concurrency control itself [Kung and Robinson 1981], but choosing an appropriate implementation is nontrivial. Like us, Reid et al. [2011] formulate the commit/abort decision for distributed transactions in the Hyder system as a function of a conflict window over a totally ordered stream of transaction intentions. Unlike us, they explicitly make a choice to use the spatial state of the system (i.e., the index) to decide transactions. A number of factors drive our choice in the opposite direction: we need to support writes at arbitrary granularity (e.g., an inode) without increasing index size; our intention log is a local in-memory array and not distributed or shared across the network, drastically reducing the size of the conflict window; and checking for conflicts using read/write-sets is easy since our index is a simple address space.

4.2. Applying Transactions

If the outcome of the decision phase is commit, *endTX* proceeds to apply the transaction to the logical address space. The first step in this process is to append the writes within the transaction to the persistent log. This step can be executed in parallel for multiple transactions, as soon as each one's decision is known, since the existence and order of writes on the log signify nothing: the multiversion index still points to older entries in the log. The second step involves changing the multiversion index to point to the new entries. Once the index has been changed, the transaction can be acknowledged and its effects are visible.

One complication is that this protocol introduces a lost update anomaly. Consider a transaction that reads a block (say, an allocation bitmap in a file system), examines and changes the first bit, and writes it back. A second transaction reads the same block concurrently, examines and changes the last bit, and writes it back. Our conflict detection scheme will correctly allow both transactions to commit. However, each transaction will write its own version of the 4KB bitmap, omitting the other's modification; as a result, the transaction with the higher timestamp will destroy the earlier transaction's modification. To avoid such lost updates, the *endTX* call performs an additional step for each transaction before appending its buffered writes to the log. Once it knows that the current transaction can commit, it scans the conflict window and *merges* updates made by prior committed transactions to the blocks in its write-set.

4.3. Implementation Details

Isotope is implemented as an in-kernel software module in Linux 2.6.38, specifically, as a device mapper that exposes multiple physical block devices as a single virtual disk, at the same level of the stack as software RAID. Next, we discuss the details of this implementation.

Log implementation: Isotope implements the log (i.e., E in Figure 3) over a conventional address space with a counter marking the tail (and additional bookkeeping information for garbage collection, which we discuss shortly). From a correctness and functionality standpoint, Isotope is agnostic to how this address space is realized. For good performance, it requires an implementation that works well for a logging workload where writes are concentrated at the tail, while reads and garbage collection can occur at random locations in the body. A naive solution is to use a single physical disk (or a RAID-0 or RAID-10 array of disks), but garbage collection activity can

hurt performance significantly by randomizing the disk arm. Replacing the disks with SSDs increases the cost-to-capacity ratio of the array without entirely eliminating the performance problem [Skourtis et al. 2014].

As a result, we use a design where a log is chained across multiple disks or SSDs (similar to Gecko [Shin et al. 2013]). Chaining the log across drives ensures that garbage collection—which occurs in the body of the log/chain—is separated from the first-class writes arriving at the tail drive of the log/chain. In addition, a commodity SSD is used as a read cache with an affinity for the tail drive of the chain, preventing application reads from disrupting write sequentiality at the tail drive. In essence, this design "collars" the throughput of the log, pegging write throughput to the speed of a single drive, but simultaneously eliminating the throughput troughs caused by concurrent garbage collection and read activity.

Garbage collection (GC): Compared to conventional log-structured stores, GC is slightly complicated in Isotope by the need to maintain older versions of blocks. Isotope tracks the oldest start timestamp across all ongoing transactions and makes a best-effort attempt to not garbage collect versions newer than this timestamp. In the worst case, any noncurrent versions can be discarded without compromising safety, by first preemptively aborting any transactions reading from them. The application can simply retry its transactions, obtaining a new, current snapshot. This behavior is particularly useful for dealing with the effects of rogue transactions that are never terminated by the application. The alternative, which we did not implement, is to set a flag that preserves a running transaction's snapshot by blocking new writes if the log runs out of space; this may be required if it's more important for a long-running transaction to finish (e.g., if it's a critical backup) than for the system to be online for writes.

Caching: The *please_cache* call in Isotope allows the application to mark the blocks it wants cached in RAM. To implement caching, Isotope annotates the multiversion index with pointers to cached copies of block versions. This call is merely a hint and provides no guarantees to the application. In practice, our implementation uses a simple LRU scheme to cache a subset of the blocks if the application requests caching indiscriminately.

Index persistence: Thus far, we have described the multiversion index as an in-memory data structure pointing to entries on the log. Changes to the index have to be made persistent so that the state of the system can be reconstructed on failures. To obtain persistence and failure atomicity for these changes, we use a *metadata log*. The size of this log can be limited via periodic checkpoints.

A simple option is to store the metadata log on battery-backed RAM or on newer technologies such as PCM or flash-backed RAM (e.g., Fusion-io's AutoCommit Memory [SanDisk 2015b]). In the absence of special hardware on our experimental testbed, we instead used a commodity SSD. Each transaction's description in the metadata log is quite compact (i.e., the logical block address and the physical log position of each write in it, and its commit timestamp). To avoid the slowdown and flash wear-out induced by logging each transaction separately as a synchronous page write, we batch multiple committed transactions together [DeWitt et al. 1984], delaying the final step of modifying the multiversion index and acknowledging the transaction to the application. We do not turn off the write cache on the SSD, relying on its ability to flush and persist data on power failures using supercapacitors.

Memory overhead: A primary source of memory overhead in Isotope is the multiversion index. A single-version index that maps a 2TB logical address space to an 4TB physical address space can be implemented as a simple array that requires 2GB of

RAM (i.e., half a billion 4-byte entries), which can be easily maintained in RAM on modern machines. Associating each address with a version (without supporting access to prior versions) doubles the space requirement to 4GB (assuming 4-byte timestamps), which is still feasible. However, multiversioned indices that allow access to past versions are more expensive, because multiple versions need to be stored, and because a more complex data structure is required instead of an array with fixed-size values. These concerns are mitigated by the fact that Isotope is not designed to be a fully fledged multiversion store; it only stores versions from the recent past, corresponding to the snapshots seen by executing transactions.

Accordingly, Isotope maintains a pair of indices: a single-version index in the form of a simple array and a multiversion index implemented as a hash table. Each entry in the single-version index contains either a valid physical address if the block has only one valid, non-GC'ed version; a null value if the block has never been written; or a constant indicating the existence of multiple versions. If a transaction issues a read and encounters this constant, the multiversion index is consulted. An address is moved from the single-version index to the multiversion index when it goes from having one version to two; it is moved back to the single-version index when its older versions are garbage collected (as described earlier in this section).

The multiversion index consists of a hash table that maps each logical address to a linked list of its existing versions, in timestamp order. Each entry contains forward and backward pointers, the logical address, the physical address, and the timestamp. A transaction walks this linked list to find the entry with the highest timestamp less than its snapshot timestamp. In addition, the entry also has a pointer to the in-memory cached copy, as described earlier. If an address is cached, the first single-version index is marked as having multiple versions even if it does not, forcing the transaction to look at the hash table index and encounter the cached copy. In the future, we plan on applying recent work on compact, concurrent maps [Fan et al. 2013] to further reduce overhead.

Rogue transactions: Another source of memory overhead in Isotope is the buffering of writes issued by in-progress transactions. Each write adds an entry to the write-set of the transaction containing the 4KB payload and a $\frac{4K}{C}$-bit bitmap, where C is the granularity of conflict detection (e.g., with 16-byte detection, the bitmap is 256 bits). Rogue transactions that issue a large number of writes are a concern, especially since transactions can be exposed to end-user applications. To handle this, Isotope provides a configuration parameter to set the maximum number of writes that can be issued by a transaction (set to 256 by default); beyond this, writes return an error code. Another parameter sets the maximum number of outstanding transactions a single process can have in-flight (also set to 256 by default). Accordingly, the maximum memory a rogue process can use within Isotope for buffered writes is roughly 256MB. When a process is killed, its outstanding transactions are preemptively aborted.

Despite these safeguards, it is still possible for Isotope to run out of memory if many processes are launched concurrently and each spams the system with rogue, never-ending transactions. In the worst case, Isotope can always relieve memory pressure by preemptively aborting transactions. Another option we considered is to flush writes to disk before they are committed; since the metadata index does not point to them, they won't be visible to other transactions. Given that the system is only expected to run out of memory in pathological cases where issuing I/O might worsen the situation, we didn't implement this scheme.

Note that the in-memory array that Isotope uses for conflict detection is not a major source of memory overhead; pointers to transaction intention records are inserted into this array in timestamp order only after the application calls *endTX*, at which point it has relinquished control to Isotope and cannot prolong the transaction. As a result,

Table I. Lines of Code for Isotope Storage Systems

Application	Original with Locks	Basic APIs (Lines Modified)	Optional APIs (Lines Added)
IsoHT	591	591 (15)	617 (26)
IsoBT	1,229	1,229 (12)	1,246 (17)
IsoFS	997	997 (19)	1,022 (25)

the lifetime of an entry in this array is short and limited to the duration of the *endTX* call.

5. ISOTOPE APPLICATIONS

To illustrate the usability and performance of Isotope, we built four applications using Isotope transactions: IsoHT, a key-value store built over a persistent hash table; IsoBT, a key-value store built over a persistent B-tree; IsoFS, a user-space POSIX file system; and ImgStore, an image storage service that stores images in IsoHT, and a secondary index in IsoBT. These applications implement each call in their respective public APIs by following a simple template that wraps the entire function in a single transaction, with a retry loop in case the transaction aborts due to a conflict (see Figure 2).

5.1. Transactional Key-Value Stores

Library-based or "embedded" key-value stores (such as LevelDB or Berkeley DB) are typically built over persistent, on-disk data structures. We built two key-value stores called IsoHT and IsoBT, implemented over an on-disk hash table and B-tree data structure, respectively. Both key-value stores support basic put/get operations on key-value pairs, while IsoBT additionally supports range queries. Each API call is implemented via a single transaction of block reads and block writes to an Isotope volume.

We implemented IsoHT and IsoBT in three stages. First, we wrote code without Isotope transactions, using a global lock to guard the entire hash table or B-tree. The resulting key-value stores are functional but slow, since all accesses are serialized by the single lock. Further, they do not provide failure atomicity: a crash in the middle of an operation can violate data structure integrity.

In the second stage, we simply replaced the acquisitions/releases on the global lock with Isotope *beginTX/endTX/abortTX* calls, without changing the overall number of lines of code. With this change, the key-value stores provide both fine-grained concurrency control (at block granularity) and failure atomicity. Finally, we added optional *mark_accessed* calls to obtain subblock concurrency control and *please_cache* calls to cache the data structures (e.g., the nodes of the B-tree, but not the values pointed to by them). Table I reports on the lines of code (LOC) counts at each stage for the two key-value stores.

Overall, Isotope APIs are simple to integrate into an existing code basis, which involved less than 50 LOC addition. Having direct support for transaction from block devices and especially not having to write failover code made the application design very simple.

5.2. Transactional File System

IsoFS is a simple user-level file system built over Isotope accessible via FUSE [fus 2016], comprising 1K lines of C code. Its on-disk layout consists of distinct regions for storing inodes, data, and an allocation bitmap for each. Each inode has an indirect pointer and a double indirect pointer, both of which point to pages allocated from the data region. Each file system call (e.g., *setattr*, *lookup*, or *unlink*) uses a single transaction to access and modify multiple blocks. The only functionality implemented

by IsoFS is the mapping and allocation of files and directories to blocks; atomicity, isolation, and durability are handled by Isotope.

IsoFS is stateless, caching neither data nor metadata across file system calls (i.e., across different transactions). Instead, IsoFS tells Isotope which blocks to cache in RAM. This idiom turned out to be surprisingly easy to use in the context of a file system; we ask Isotope to cache all bitmap blocks on startup, each inode block when an inode within it is allocated, and each data block that's allocated as an indirect or double indirect block. Like the key-value stores, IsoFS was implemented in three stages and required few extra lines of code to go from a global lock to using the Isotope API (see Table I).

IsoFS trivially exposes transactions to end applications over files and directories. For example, a user might create a directory, move a file into it, edit the file, and rename the directory, only to abort the entire transaction and revert the file system to its earlier state. One implementation-related caveat is that we were unable to expose transactions to end applications of IsoFS via the FUSE interface, since FUSE decouples application threading from file system threading and does not provide any facility for explicitly transferring a transaction handle on each call. Accordingly, we can only expose transactions to the end application if IsoFS is used directly as a library within the application's process.

5.3. Experience

Composability: As we stated earlier, Isotope-based storage systems are trivially composable: a single transaction can encapsulate calls to IsoFS, IsoHT, and IsoBT. To demonstrate the power of such composability, we built ImgStore, the image storage application described in Section 3. ImgStore stores images in IsoHT, using 64-bit IDs as keys. It then stores a secondary index in IsoBT, mapping dates to IDs. The implementation of ImgStore is trivially simple: to add an image, it creates a transaction to put the image in IsoHT, and then updates the secondary index in IsoBT. The result is a storage system that—in just 148 LOC—provides hash-table-like performance for gets while supporting range queries.

Isolation levels: Isotope provides both strict serializability and snapshot isolation; our expectation was that developers would find it difficult to deal with the semantics of the latter. However, our experience with IsoFS, IsoHT, and IsoBT showed otherwise. Snapshot isolation provides better performance than strict serializability but introduces the write skew anomaly [Berenson et al. 1995]: if two concurrent transactions read two blocks and each updates one of the blocks (but not the same one), they will both commit despite not being serializable in any order. The write skew anomaly is problematic for applications if a transaction is expected to maintain an integrity constraint that includes some block it does not write to (e.g., if the two blocks in the example have to sum to less than some constant). In the case of the storage systems we built, we did not encounter these kinds of constraints; for instance, no particular constraint holds between different bits on an allocation map. As a result, we found it relatively easy to reason about and rule out the write skew anomaly.

Randomization: Our initial implementations exhibited a high abort rate due to deterministic behavior across different transactions. For example, a simple algorithm for allocating a free page involved getting the first free bit from the allocation bitmap; as a result, multiple concurrent transactions interfered with each other by trying to allocate the same page. To reduce the abort rate, it was sufficient to remove the determinism in simple ways; for example, we assigned each thread a random start offset into the allocation bitmap.

6. PERFORMANCE EVALUATION

We evaluate Isotope on a machine with an Intel Xeon CPU with 24 hyperthreaded cores, 24GB RAM, three 10K RPM disks of 600GB each, an 128GB SSD for the OS, and two other 240GB SSDs with SATA interfaces. In the following experiments, we used two primary configurations for Isotope's persistent log: a three-disk chained logging instance with a 32GB SSD read cache in front, and a two-SSD chained logging instance. In some of the experiments, we compare against conventional systems running over RAID-0 configurations of three disks and two SSDs, respectively. In the chained logging configurations, all writes are logged to the single tail drive, while reads are mostly served by the other drives (and the SSD read cache for the disk-based configuration). The performance of this logging design under various workloads and during GC activity has been documented in Shin et al. [2013]. In all our experiments, GC is running in the background and issuing I/Os to the drives in the body of the chain to compact segments, without disrupting the tail drive.

In this section, we first focus on the performance and overhead of Isotope, showing that it exploits fine-grained concurrency in workloads and provides high, stable throughput. Then we show that Isotope applications—in addition to being simple and robust—are fast, efficient, and composable into larger applications.

6.1. Isotope Performance

To understand how Isotope performs depending on the concurrency present in the workload, we implemented a synthetic benchmark. The benchmark executes a simple type of transaction that reads three randomly chosen blocks, modifies a random 16-byte segment within each block (aligned on a 16-byte boundary), and writes them back. This benchmark performs identically with strict serializability and snapshot isolation, since the read-set exactly matches the write-set.

In the following experiments, we executed 64 instances of the microbenchmark concurrently, varying the size of the address space accessed by the instances to vary contention. The blocks are chosen from a specific prefix of the address space, which is a parameter to the benchmark; the longer this prefix is, the bigger the fraction of the address space accessed by the benchmark, and the less skewed the workload. The two key metrics of interest are transaction goodput (measured as the number of successfully committed transactions per second, as well as the total number of bytes read or written per second by these transactions) and overall transaction throughput; their ratio is the commit rate of the system. Each data point in the following graphs is averaged across three runs; in all cases, the minimum and the maximum run were within 10% of the average.

Figure 5 shows the performance of this benchmark against Isotope without fine-grained conflict detection; that is, the benchmark does not issue *mark_accessed* calls for the 16-byte segments it modifies. On the x-axis, we increase the fraction of the address space accessed by the benchmark. On the left y-axis, we plot the rate at which data is read and written by transactions; on the right y-axis, we plot the number of transactions/second. On both disk and SSD, transactional contention cripples performance on the left part of the graph: even though the benchmark attempts to commit thousands of transactions/second, all of them access a small number of blocks, leading to low goodput. Note that overall transaction throughput is very high when the commit rate is low: aborts are cheap and do not result in storage I/O.

Conversely, disk contention hurts performance on the right side of Figure 5-Left: since the blocks read by each transaction are distributed widely across the address space, the 32GB SSD read cache is ineffective in serving reads and the disk arm is randomized and seeking constantly. As a result, the system provides very few transactions per second

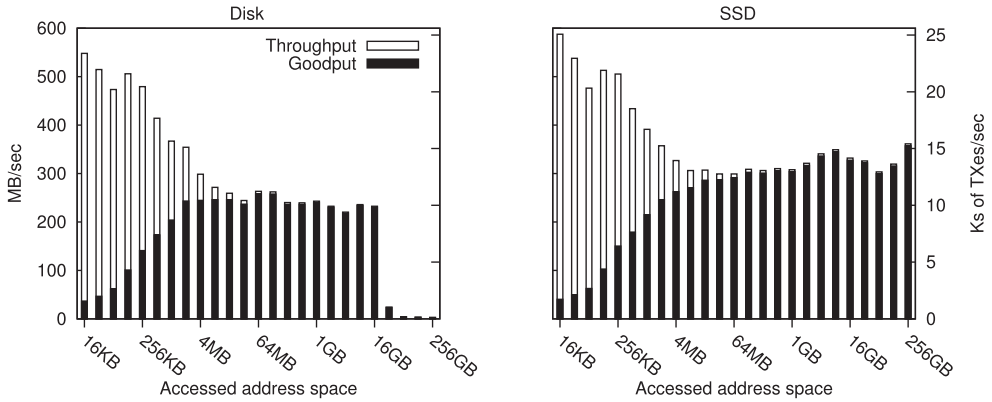

Fig. 5. Without fine-grained conflict detection, Isotope performs well under low contention.

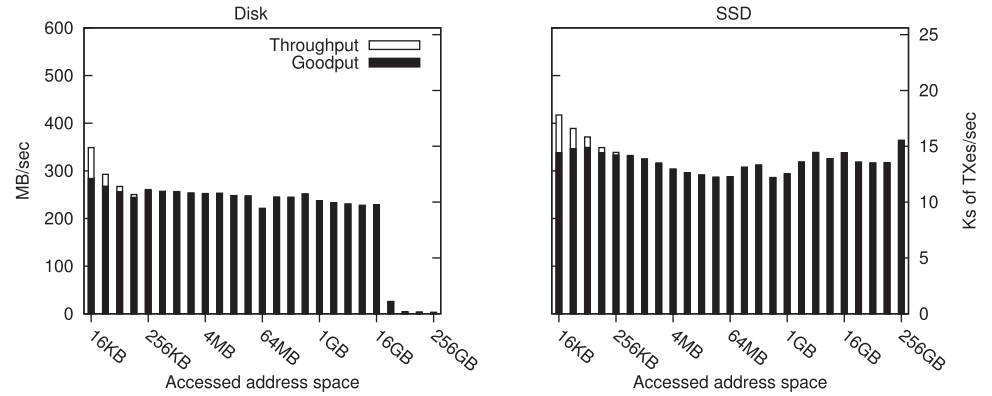

Fig. 6. With fine-grained conflict detection, Isotope performs well even under high contention.

(though with a high commit rate). In the middle of the graph is a sweet spot where Isotope saturates the disk at roughly 120MB/s of writes, where the blocks accessed are concentrated enough for reads to be cacheable in the SSD (which supplies 120MB/s of reads, or 30K 4KB IOPS), while distributed enough for writes to not trigger frequent conflicts. However, Isotope running on top of SSDs, which are less affected by random reads, constantly saturates the throughput (Figure 5-Right).

We can improve performance on the left side of the graphs in Figure 5 via fine-grained conflict detection. In Figure 6, the benchmark issues *mark_accessed* calls to tell Isotope which 16-byte fragment it is modifying. The result is high, stable goodput even when all transactions are accessing a small number of blocks, since there is enough fragment-level concurrency in the system to ensure a high commit rate. Isotope's conflict detection was not CPU intensive: we observed an average CPU utilization of 5.96% without fine-grained conflict detection, and 6.17% with it.

Figure 7 shows the ability of Isotope to exploit different levels of fine-grained concurrency in the workload by choosing an appropriate conflict detection granularity smaller than a block size. We access a small, 32KB address space with the synthetic benchmark and vary the size of the fragment updated by each transaction, as well as the granularity of conflict detection. As the fragments become smaller on the x-axis, goodput goes up due to more concurrency in the workload.

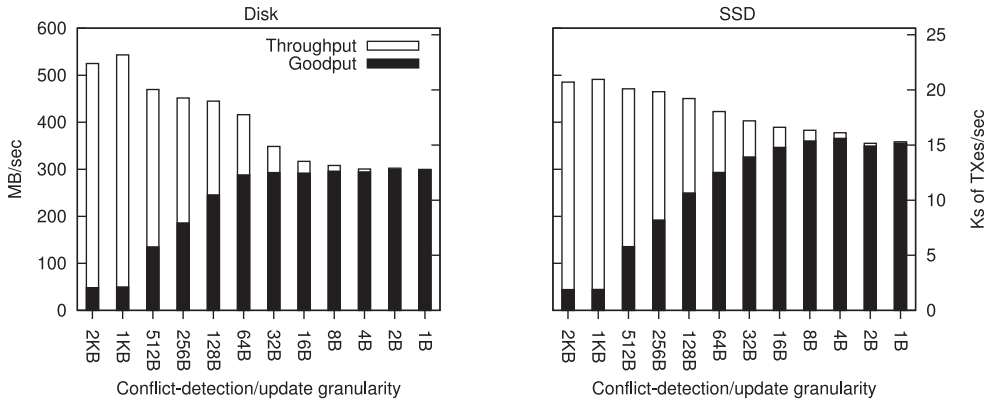

Fig. 7. Isotope provides higher goodput as it detects conflicts at finer grain.

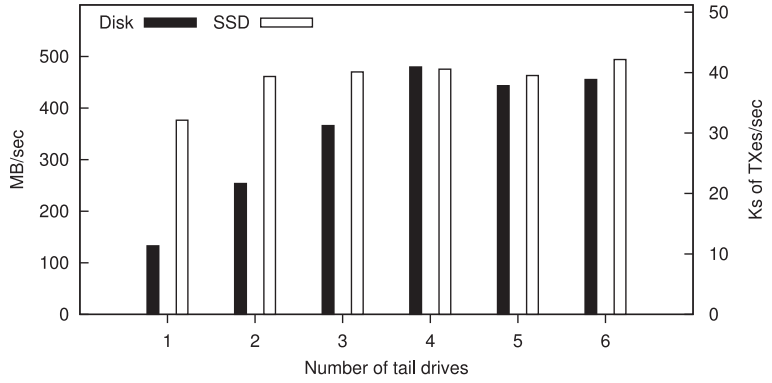

Fig. 8. Isotope scalability to larger disk arrays.

Finally, we evaluated Isotope's scalability to larger arrays in Figure 8. For this experiment, we changed the benchmark's transactions to always read from a small address space completely cached in RAM (using *please_cache*) and issued blind writes to random locations in the entire address space; we did this to remove our SSD cache as a read bottleneck. We then used a single RAID-0 volume of multiple drives as the logging chain's tail drive. The figure only contains the performance measured from the block device and not from the in-memory cache. As seen, throughput scales up to 500MB/s with three to four disk drives or two SSD drives before plateauing; at this speed, the SSD we use to log metadata updates becomes the bottleneck.

6.2. Key-Value Store Performance

As described earlier, we implemented two key-value stores over Isotope: IsoHT using a hash table index and IsoBT using a B-tree index, respectively. IsoBT exposes a fully functional LevelDB API to end applications; IsoHT does the same minus range queries. To evaluate these systems, we used the LevelDB benchmark [Google 2016] as well as the YCSB [Cooper et al. 2010] benchmark. We ran the fill-random, read-random, and delete-random workloads of the LevelDB benchmark and YCSB workload-A traces (50% reads and 50% updates following a zipf distribution on keys). All these experiments are on the two-SSD configuration of Isotope. For comparison, we ran LevelDB on a RAID-0 array of the two SSDs, in both synchronous mode ("LvlDB-s")

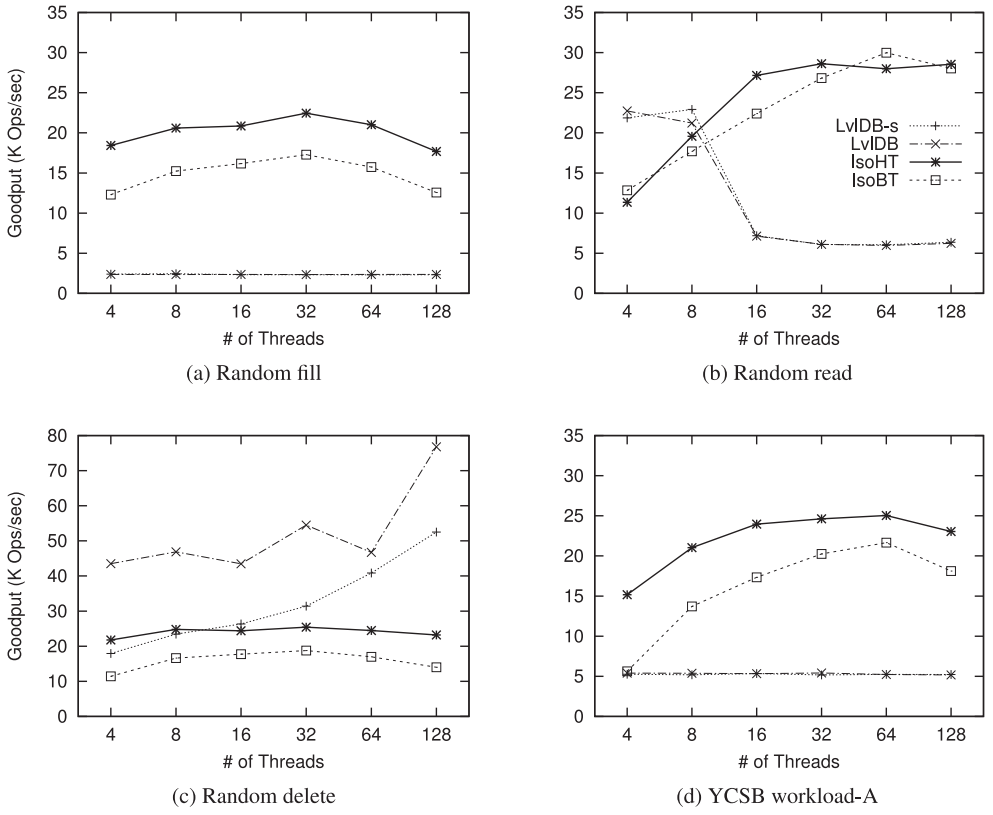

Fig. 9. IsoHT and IsoBT outperform LevelDB for data operations while providing stronger consistency guarantees.

and asynchronous mode ("LvlDB"). LevelDB was set to use no compression and the default write cache size of 8MB. For all the workloads, we used a value size of 8KB and varied the number of threads issuing requests from four to 128. Results with different value sizes (from 4KB to 32KB) showed similar trends.

For operations involving writes (Figures 9(a), 9(c), and 9(d)), IsoHT and IsoBT goodput increases with the number of threads but dips slightly beyond 64 threads due to an increased transaction conflict rate. For the read workload (Figure 9(b)), throughput increases until the underlying SSDs are saturated. Overall, IsoHT has higher goodput than IsoBT, since it touches fewer metadata blocks per operation. We ran these experiments with Isotope providing snapshot isolation, since it performed better for certain workloads and gave sufficiently strong semantics for building the key-value stores. We compare strict serializability and snapshot isolation in the next subsection.

LevelDB's performance is low for fill operations due to sorting and multilevel merging (Figure 9(a)), and its read performance degrades as the number of concurrent threads increases because of the CPU contention in the skip list, cache thrashing, and internal merging operations (Figure 9(b)). Still, LevelDB's delete is very efficient because it only involves appending a small delete intention record to a log, whereas IsoBT/IsoHT has to update a full 4KB block per delete (Figure 9(c)).

The point of this experiment is not to show that IsoHT/IsoBT is better than LevelDB, which has a different internal design and is optimized for specific workloads such as sequential reads and bulk writes. Rather, it shows that systems built over Isotope with

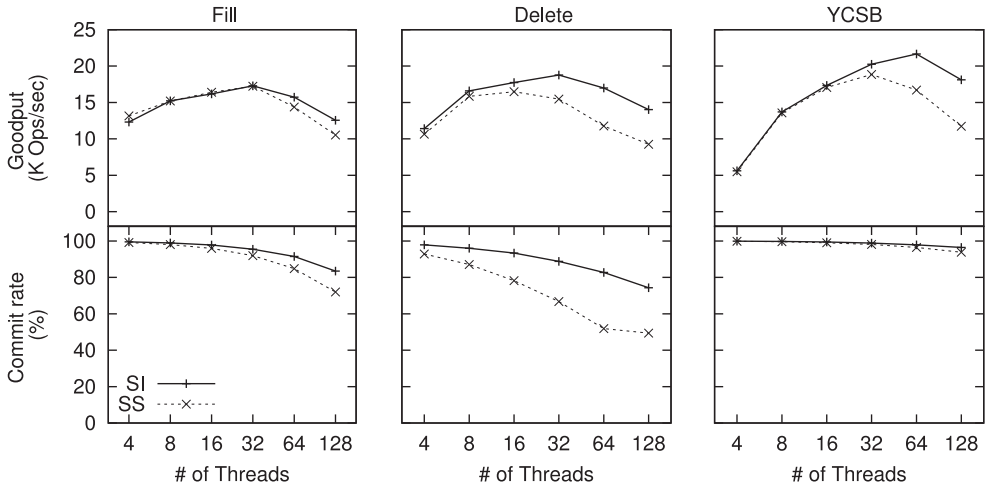

Fig. 10. Goodput and commit rate of IsoBT using snapshot isolation and strict serializability.

little effort can provide equivalent or better performance than an existing system that implements its own concurrency control and failure atomicity logic.

6.3. Snapshot Isolation Versus Strict Serializability

Isotope supports snapshot isolation and strict serializability. Using one or the other does not incur extra overhead, because the only difference between the two is whether to check write-write or read-write conflicts among transactions. However, the transaction commit rate that applications observe varies depending on the semantics. Figure 10 shows the goodput and the transaction commit rate of IsoBT using the same experimental setup as the previous key-value store experiments running over Isotope, but using snapshot isolation (SI) and strict serializability (SS). The LevelDB benchmark with random-read is not used, because a workload without writes works the same under both semantics. We only show the case of IsoBT because IsoHT and IsoFS have inherently lower transactional conflict rates due to their different metadata structures.

The figure shows that snapshot isolation leads to better performance than strict serializability in general. The goodput measured from the two semantics does not diverge when there is a small number of threads. However, when the concurrency increases, strict serializability displays a lower commit rate and lower goodput compared to snapshot isolation. For the fill workload, snapshot isolation leads to 10% better performance than strict serializability until 64 threads are used and 16% better when the number of threads reaches 128. Delete operations show a higher variance of performance and commit rate, because there are only metadata operations, where transactions are concentrated on a small number of metadata blocks with no data accesses. The YCSB workload shows a small variance of goodput under low concurrency, but once the number of threads reaches 64 or above, small differences of commit rates lead to a relatively large gap of goodput. The commit rate for YCSB is high overall, because 50% of requests are reads, and the other 50%, which are writes, make the goodput vary.

This experiment shows the tradeoff between strong transactional semantics and performance: if one needs stronger guarantees, performance needs to be sacrificed. Isotope provides both strong and slightly weaker semantics depending on the user needs. As snapshot isolation performs better and provides strong enough semantics to maintain IsoBT, IsoHT, and IsoFS in a consistent state, we use snapshot isolation for the rest of the evaluation.

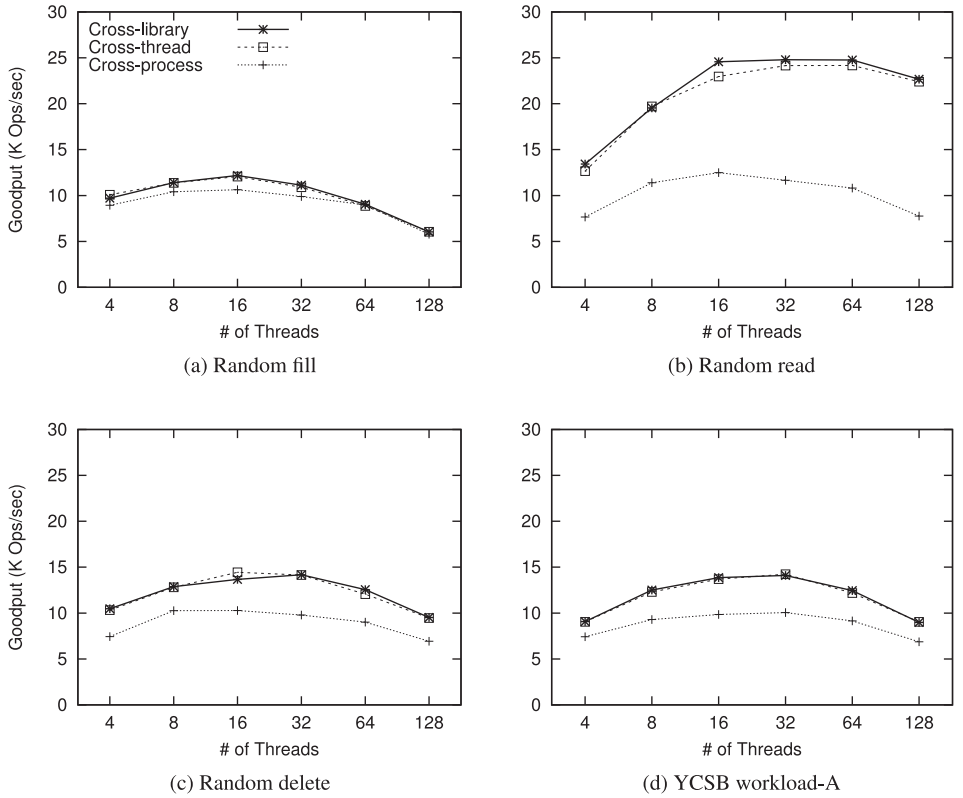

Fig. 11. ImgStore performance under different compositions of IsoBT and IsoHT.

6.4. Composability

To evaluate the composability of Isotope-based storage systems, we ran the same experiment as the key-value store evaluation on ImgStore, our image storage application built over IsoHT and IsoBT. In this experiment, ImgStore transactionally stores a 16KB payload (corresponding to an image) in IsoHT and a small date-to-ID mapping in IsoBT. To capture the various ways in which Isotope storage systems can be composed (see Section 3), we implemented several versions of ImgStore: cross-library, where ImgStore accesses the two key-value stores as in-process libraries, with each transaction executing within a single user-space thread; cross-thread, where ImgStore accesses each key-value store using a separate thread and requires transactions to execute across them; and cross-process, where each key-value store executes within its own process and is accessed by ImgStore via socket-based IPC. Figure 11 shows the resulting performance for all three versions.

The performance trend observed in each workload in the figure is similar to the IsoHT and IsoBT; the performance increases as the number of concurrent threads increases and plateaus or decreases after a certain level of concurrency is reached. ImgStore exhibits less concurrency and goodput than IsoHT or IsoBT (peaking at 16 to 32 threads), since each composite transaction conflicts if either of its constituent transactions in underlying IsoHT or IsoBT conflict.

The comparison between cross-library and cross-thread shows that the cost of the extra *takeoverTX*/*releaseTX* calls required for cross-thread transactions is negligible.

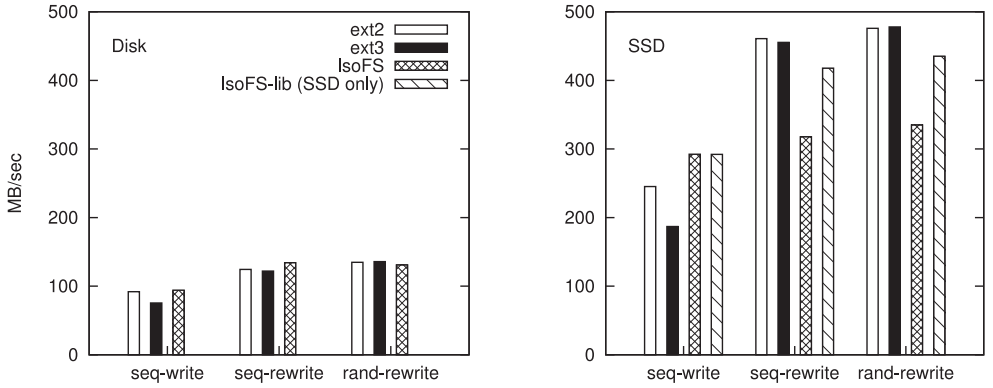

Fig. 12. IOZone over IsoFS and ext2/ext3.

For all benchmarks, cross-process transactions are the slowest due to the extra IPC overhead.

6.5. File System Performance

Next, we compare the end-to-end performance of IsoFS running over Isotope using the IOZone [IOzone 2016] write/rewrite benchmark with eight threads. Each thread writes to its own file using a 16KB record size until the file size reaches 256MB; it then rewrites the entire file sequentially and then rewrites it randomly. We ran this workload against IsoFS running over Isotope, which converted each 16KB write into a transaction involving four 4KB Isotope writes, along with metadata writes. We also ran ext2 and ext3 over Isotope; these issued solitary, nontransactional reads and writes, which were interpreted by Isotope as singleton transactions (in effect, Isotope operated as a conventional log-structured block store, so that ext2 and ext3 are not penalized for random I/Os). We ran ext3 in "ordered" mode, where metadata is journaled but file contents are not.

Figure 12 plots the throughput observed by IOZone: on disk, IsoFS matches or slightly outperforms ext2 and ext3, saturating the tail disk on the chain. On SSD, IsoFS is faster than ext2 and ext3 for initial writes, but is bottlenecked by FUSE on rewrites. When we ran IsoFS directly using a user-space benchmark that mimics IOZone ("IsoFS-lib"), throughput improved to over 415MB/s. A secondary point made by this graph is that Isotope does not slow down applications that do not use its transactional features (the high performance is mainly due to the underlying logging scheme, but ext2 and ext3 still saturate disk and SSD for rewrites), satisfying a key condition for pushing functionality down the stack [Saltzer et al. 1984].

7. RELATED WORK

The idea of transactional atomicity for multiblock writes was first proposed in Mime [Chao et al. 1992], a log-structured storage system that provided atomic multisector writes. Over the years, multiple other projects have proposed block-level or page-level atomicity: the Logical Disk [De Jonge et al. 1993] in 1993, Stasis [Sears and Brewer 2006] in 2006, TxFlash [Prabhakaran et al. 2008] in 2008, and MARS [Coburn et al. 2013] in 2013. RVM [Satyanarayanan et al. 1994] and Rio Vista [Lowell and Chen 1997] proposed atomicity over a persistent memory abstraction. All these systems explicitly stopped short of providing full transactional semantics, relying on higher layers to implement isolation. To the best of our knowledge, no existing single-machine system

has implemented transactional isolation at the block level, or indeed any concurrency control guarantee beyond linearizability.

On the other hand, distributed file systems have often relied on the underlying storage layer to provide concurrency control. Boxwood [MacCormick et al. 2004], Sinfonia [Aguilera et al. 2007], and CalvinFS [Thomson and Abadi 2015] presented simple NFS designs that leveraged transactions over distributed implementations of high-level data structures and a shared address space. Transaction isolation has been proposed for shared block storage accessed over a network [Amiri et al. 2000] and for key-value stores [Sovran et al. 2011]. Isotope can be viewed as an extension of similar ideas to single-machine, multicore systems that does not require consensus or distributed rollback protocols. Our single-machine IsoFS implementation has much in common with the Boxwood, Sinfonia, and CalvinFS NFS implementations that ran against clusters of storage servers.

Isotope also fits into a larger body of work on smart single-machine block devices, starting with Loge [English and Stepanov 1992] and including HP AutoRAID [Wilkes et al. 1996]. Some of this work has focused on making block devices smarter without changing the interface [Sivathanu et al. 2003], while other work has looked at augmenting the block interface [Chao et al. 1992; Wang et al. 1998; Ganger 2001], modifying it [Zhang et al. 2012], and even replacing it with an object-based interface [Mesnier et al. 2003]. In a distributed context, Parallax [Meyer et al. 2008] and Strata [Cully et al. 2014] provide virtual disks on storage clusters. A number of file systems are multiversion, starting with WAFL [Hitz et al. 1994], and including many others [Santry et al. 1999; Muniswamy-Reddy et al. 2004; Cornell et al. 2004]. Underlying these systems is research on multiversion data structures [Driscoll et al. 1986]. Less common are multiversion block stores such as Clotho [Flouris and Bilas 2004] and Venti [Quinlan and Dorward 2002].

A number of file systems have been built over a full-fledged database. Inversion [Olson 1993] is a conventional file system built over the POSTGRES database, while Amino [Wright et al. 2007] is a transactional file system (i.e., exposing transactions to users) built over Berkeley DB. WinFS [Microsoft 2016b] was built over a relational engine derived from the SQL Server. This route requires storage system developers to adopt a complex interface—one that does not match or expose the underlying grain of the hardware—in order to obtain benefits such as isolation and atomicity. In contrast, Isotope retains the simple block storage interface while providing isolation and atomicity.

TxOS [Porter et al. 2009] is a transactional operating system that provides ACID semantics over syscalls including file accesses. In contrast, Isotope is largely OS agnostic and can be ported easily to commodity operating systems, or even implemented under the OS as a hardware device. In addition, Isotope supports the easy creation of new systems such as key-value stores and file systems that run directly over block storage.

Isotope is also related to the large body of work on software transactional memory (STM) [Shavit and Touitou 1997; Harris et al. 2010] systems, which typically provide isolation but not durability or atomicity. Recent work has leveraged new NVRAM technologies to add durability to the STM abstraction: Mnemosyne [Volos et al. 2011] and NV-Heaps [Coburn et al. 2011] with PCM and Hathi [Saxena et al. 2012a] with commodity SSDs. In contrast, Isotope aims for transactional secondary storage, rather than transactional main memory.

8. CONCLUSION

We described Isotope, a transactional block store that provides isolation in addition to atomicity and durability. We showed that isolation can be implemented efficiently within the block layer, leveraging the inherent multiversioning of log-structured block

stores and application-provided hints for fine-grained conflict detection. Isotope-based systems are simple and fast, while obtaining database-strength guarantees on failure atomicity, durability, and consistency. They are also composable, allowing application-initiated transactions to span multiple storage systems and different abstractions such as files and key-value pairs.

AVAILABILITY

The code of Isotope is available at http://gecko.cs.cornell.edu.

REFERENCES

Abutalib Aghayev and Peter Desnoyers. 2015. Skylight—a window on shingled disk operation. In *USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 135–149.

Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: A new paradigm for building scalable distributed systems. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 159–174.

Khalil Amiri, Garth A. Gibson, and Richard Golding. 2000. Highly concurrent shared storage. In *International Conference on Distributed Computing Systems*. IEEE, 298–307.

Anirudh Badam and Vivek S. Pai. 2011. SSDAlloc: Hybrid SSD/RAM memory management made easy. In *USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, 211–224.

Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A shared log design for flash clusters. In *USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, 1–14.

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Vol. 370. Addison-Wesley, New York.

Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. 1992. *Mime: A High Performance Parallel Storage Device with Strong Recovery Guarantees*. Technical Report. HPL-CSP-92-9, Hewlett-Packard Laboratories.

Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. 2013. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, 197–212.

Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 105–118.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC'10)*. ACM, 143–154.

Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante. 2004. Wayback: A user-level versioning file system for Linux. In *USENIX Annual Technical Conference (ATC'04)*. USENIX Association, 19–28.

Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffrey Lefebvre, Daniel Ferstay, and Andrew Warfield. 2014. Strata: Scalable high-performance storage on virtualized non-volatile memory. In *USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX Association, 17–31.

Wiebren De Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. 1993. The logical disk: A new approach to improving file systems. *ACM SIGOPS Operating Systems Review* 27, 5 (1993), 15–28.

David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. 1984. Implementation techniques for main memory database systems. In *ACM SIGMOD International Conference on Management of Data*. ACM, 1–8.

James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. 1986. Making data structures persistent. In *ACM Symposium on Theory of Computing (STOC'86)*. ACM, 109–121.

Robert M. English and Alexander A. Stepanov. 1992. Loge: A self-organizing disk controller. In *USENIX Winter Technical Conference*. USENIX Association, 237–251.

Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, 371–384.

fcntl(2) Linux manual page. 2016. fcntl(2) Linux manual page. Retrieved from http://man7.org/linux/man-pages/man2/fcntl.2.html.

Filesystem in Userspace. 2016. Retrieved from https://github.com/libfuse/libfuse.

Michail Flouris and Angelos Bilas. 2004. Clotho: Transparent data versioning at the block I/O level. In *IEEE Conference on Mass Storage Systems and Technologies (MSST'04)*. IEEE, 315–328.

Fusion-io. 2015. Fusion-io. Retrieved from http://www.fusionio.com.

Gregory R. Ganger. 2001. *Blurring the Line Between OSes and Storage Devices*. School of Computer Science, Carnegie Mellon University.

Google. 2016. LevelDB benchmarks. Retrieved from https://github.com/google/leveldb/blob/master/doc/benchmark.html.

Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*. ACM, 175–184.

Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory*. Morgan and Claypool Publishers.

Dave Hitz, James Lau, and Michael Malcolm. 1994. File system design for an NFS file server appliance. In *USENIX Winter Technical Conference*. USENIX Association, 235–246.

IOzone. 2016. IOzone filesystem benchmark. Retrieved from http://www.iozone.org.

Jithin Jose, Mohammad Banikazemi, Wendy Belluomini, Chet Murthy, and Dhabaleswar K Panda. 2013. MetaData persistence using storage class memory: Experiences with flash-backed DRAM. In *Proceedings of Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW'13)*. ACM, 3:1–3:7.

Hsiang-Tsung Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

David E. Lowell and Peter M. Chen. 1997. Free transactions with rio vista. *ACM SIGOPS Operating Systems Review* 31, 5 (1997), 92–101.

John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. 2004. Boxwood: Abstractions as the foundation for storage infrastructure. In *USENIX Symposium on Opearting Systems Design and Implementation (OSDI'04)*. USENIX Association, 105–120.

Mike Mesnier, Gregory R. Ganger, and Erik Riedel. 2003. Object-based storage. *IEEE Communications Magazine* 41, 8 (2003), 84–90.

Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. 2008. Parallax: Virtual disks for virtual machines. *ACM SIGOPS Operating Systems Review* 42, 4 (2008), 41–54.

Microsoft. 2016a. Storage Spaces. Retrieved from http://technet.microsoft.com/en-us/library/hh831739.aspx.

Microsoft. 2016b. WinFS. Retrieved from http://blogs.msdn.com/b/winfs/.

C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. 2004. A versatile and user-oriented versioning file system. In *USENIX Conference on File and Storage Technologies (FAST'04)*. USENIX Association, 115–128.

Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. 2012. Flat datacenter storage. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, 1–15.

Michael A. Olson. 1993. The design and implementation of the inversion file system. In *USENIX Winter Technical Conference*. USENIX Association, 205–218.

Avery Pennarun. 2016. Everything you never wanted to know about file locking. Retrieved from http://apenwarr.ca/log/?m=201012#13.

Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. 2009. Operating system transactions. In *ACM Symposium on Operating Systems Principles (SOSP'09)*. ACM, 161–176.

Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. 2008. Transactional flash. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, 147–160.

Sean Quinlan and Sean Dorward. 2002. Venti: A new approach to archival storage. In *USENIX Conference on File and Storage Technologies (FAST'02)*. USENIX Association, 89–101.

Colin Reid, Philip A. Bernstein, Ming Wu, and Xinhao Yuan. 2011. Optimistic concurrency control by melding trees. *Proceedings of the VLDB Endowment* 4, 11 (2011).

Jerome H. Saltzer, David P. Reed, and David D. Clark. 1984. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)* 2, 4 (1984), 277–288.

SanDisk. 2015a. SanDisk Fusion-io Atomic Multi-Block Writes. Retrieved from http://www.sandisk.com/assets/docs/accelerate-myql-open-source-databases-with-sandisk-nvmfs-and-fusion-iomemory-sx300-application-accelerators.pdf.

SanDisk. 2015b. SanDisk Fusion-io Auto-Commit Memory. Retrieved from http://web.sandisk.com/assets/white-papers/MySQL_High-Speed_Transaction_Logg ing.pdf.

Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. 1999. Deciding when to forget in the elephant file system. *ACM SIGOPS Operating Systems Review* 33, 5 (1999), 110–123.

Mahadev Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. 1994. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems (TOCS)* 12, 1 (1994), 33–57.

Mohit Saxena, Mehul A. Shah, Stavros Harizopoulos, Michael M. Swift, and Arif Merchant. 2012a. Hathi: Durable transactions for memory using flash. In *International Workshop on Data Management on New Hardware*. ACM, 33–38.

Mohit Saxena, Michael M. Swift, and Yiying Zhang. 2012b. FlashTier: A lightweight, consistent and durable storage cache. In *ACM European Conference on Computer Systems (EuroSys'12)*. ACM, 267–280.

Seagate. 2016. Seagate Kinetic Open Storage Platform. Retrieved from http://www.seagate.com/solutions/cloud/data-center-cloud/platforms/.

Russell Sears and Eric Brewer. 2006. Stasis: Flexible transactional storage. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, 29–44.

Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (1997), 99–116.

Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. 2013. Gecko: Contention-oblivious disk arrays for cloud storage. In *USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX Association, 213–225.

Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2003. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies (FAST'03)*. USENIX Association, 73–88.

Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. 2014. Flash on rails: Consistent flash performance through redundancy. In *USENIX Annual Technical Conference (ATC'14)*. USENIX Association, 463–474.

Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. 2010. Extending SSD lifetimes with disk-based write caches. In *USENIX Conference on File and Storage Technologies (FAST'10)*. USENIX Association, 101–114.

Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, 385–400.

Lex Stein. 2005. Stupid file systems are better. In *Workshop on Hot Topics in Operating Systems (HotOS'05)*. USENIX Association.

Alexander Thomson and Daniel J. Abadi. 2015. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. In *USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 1–14.

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.

Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. 1998. Virtual log based file systems for a programmable disk. *Operating Systems Review* 33 (1998), 29–44.

John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. 1996. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)* 14, 1 (1996), 108–136.

Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. 2007. Extending ACID semantics to the file system. *ACM Transactions on Storage (TOS)* 3, 2 (2007), 4.

Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. De-indirection for flash-based SSDs with nameless writes. In *USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, 1–16.

Received September 2016; accepted December 2016