Exploring Storage Class Memory with Key Value Stores

Katelin A. Bailey Peter Hornyack

Luis Ceze Steven D. Gribble Henry M. Levy

University of Washington
Dept. of Computer Science and Engineering
Seattle, WA, USA
{katelin,pjh,luisceze,gribble,levy}@cs.washington.edu

ABSTRACT

In the near future, new *storage-class memory* (SCM) technologies – such as phase-change memory and memristors – will radically change the nature of long-term storage. These devices will be cheap, non-volatile, byte addressable, and near DRAM density and speed. While SCM offers enormous opportunities, profiting from them will require new storage systems specifically designed for SCM's properties.

This paper presents *Echo*, a persistent key-value storage system designed to leverage the advantages and address the challenges of SCM. The goals of Echo include high performance for both small and large data objects, recoverability after failure, and scalability on multicore systems. Echo achieves its goals through the use of a two-level memory design targeted for memory systems containing both DRAM and SCM, exploitation of SCM's byte addressability for fine-grained transactions in non-volatile memory, and the use of snapshot isolation for concurrency, consistency, and versioning. Our evaluation demonstrates that Echo's SCM-centric design achieves the durability guarantees of the best disk-based stores with the performance characteristics approaching the best in-memory key-value stores.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management

General Terms

Design, Performance, Reliability

INFLOW'13, November 3, 2013, Pennsylvania, USA.

Keywords

Operating Systems, Non Volatile Memory, storage system, main memory, key-value store

1. INTRODUCTION

For most of its history, OS data management has been severely constrained by the physical characteristics of the slow, block-oriented disks required for persistent storage. In the near future, however, emerging *storage-class memory* (SCM) technologies will likely revolutionize persistent storage. SCM devices – such as phase-change memory (PCM) [23, 39] and memristors [42] – will offer exciting and inexpensive storage options that combine the best characteristics of DRAM and disk in novel ways. The primary benefits of SCM include: (1) durability of writes across power failures, (2) density competitive with DRAM, likely to reach a terabyte on the memory bus in a few years, (3) byte-addressability, instead of the block-oriented accesses of SSD and disk, and (4) latency within two- to four-times that of DRAM access.

SCM has several weaknesses as well, however, including an asymmetry of read and write costs (with writes significantly more expensive), and write wear-out over time, which suggests the need for wear leveling. However, its speed and byte-addressable access granularity are far better than for SSDs, and even its write endurance is significantly improved.

While today's storage systems could simply substitute SCM for current storage technologies, this approach has numerous drawbacks. For example, just replacing disks and SSDs with SCM will result in unnecessary overhead from the block-oriented structure and interface of existing file systems. Similarly, replacing all of main memory (DRAM) with SCM – without considering atomicity, for example – will result in inconsistent data on recovery after failure, in addition to a loss of performance. Therefore, technology replacement alone cannot achieve the full potential of storage-class memories. In the near future, we believe that SCM technology can significantly improve OS storage and management of persistent data, but this will require new storage systems *explicitly* designed for the characteristics of SCM.

Our work focuses on future systems that are likely to provide SCM storage directly on the memory bus *along with* DRAM; others have considered this environment as well ([10, 44]). In

that context, this paper describes the high-level architecture and implementation of a storage system designed to exploit the advantages, and to mitigate the challenges, of future SCM-based technologies in such architectures. Called *Echo*, our system has three main design goals:

1. **Variable granularity.** As a replacement for both main memory and disk stores, Echo should efficiently manage storage and retrieval of both small and large data blocks – from a few bytes to many megabytes.
2. **Consistency and recoverability.** Echo must preserve data consistency and avoid corruption of durable in-memory data structures in the case of failure and restart.
3. **Scalability and concurrency.** Performance must scale with the number of clients and requests; that is, Echo must handle parallel operations over the store on increasingly large-scale multicore CPUs.

We designed Echo to be a lightweight, persistent, key-value storage system that uses an SCM-aware architecture to meet these goals in multiple ways. First, it relies on SCM's byte addressability to support efficient storage of both fine- and large-grained data objects. Second, it employs a two-level "hybrid" memory design, using DRAM as a thin layer to offset the performance and wear-out weaknesses of SCM. And third, it employs transactions and snapshot isolation [4] to support concurrency, data consistency, recoverability, and scalability for multicore processors.

The existence of snapshot isolation and byte addressability in non-volatile memory have convenient benefits. Snapshot isolation creates multiple *versions* of data objects to ensure that each concurrent thread sees a consistent data view. With byte addressability, Echo can cheaply and atomically commit a new object version using a single memory write (e.g., a pointer flip) to non-volatile SCM. This leads naturally to historical versioning; by simply maintaining data versions committed to non-volatile memory, Echo can easily support common versioning applications, such as auditing, backup, intrusion detection, and error recovery (e.g., [8, 16, 17, 21, 27, 30]). And by writing new object versions rather than overwriting existing ones, Echo provides a simple form of wear leveling for SCM.

Others have noted that new SCM memory technologies have the potential to fundamentally change OS design and structure [3]; e.g., removing the 50-year-old premise of a two-level store has important implications for the file system, the virtual memory, the I/O system, and so on. In this context, we believe that a general-purpose SCM-centric, key-value storage system such as Echo is a first building block in that direction, on top of which either OS or higher-level software could be crafted. While previous SCM research has focused on non-volatile persistent objects [10], non-volatile memory regions [44], and traditional file systems [11], to our knowledge, Echo is the first fine-grained, persistent key-value store specifically designed for SCM technology.

Overall, while the individual techniques used in Echo are not new in themselves, Echo provides a careful blending of in-memory and durable storage management techniques, leading to a new system explicitly tuned for the benefits of non-volatile memory technologies expected in the near future.

2. ARCHITECTURE

This section briefly describes Echo's high-level architecture, which seeks to exploit the advantages and offset the weaknesses of SCM. Specifically, the architecture strives to: (1) improve performance and simplify recoverability by clearly separating transient, uncommitted state from durable, consistent state; (2) maximize scalability through concurrency and reduced locking, and (3) expose historical versions of data to applications, while letting them choose how versions are managed.

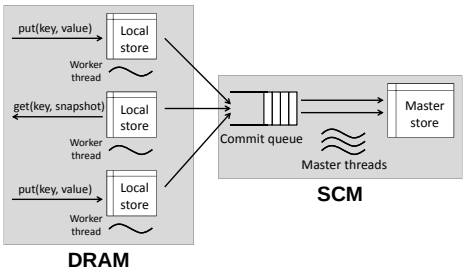

Figure 1: The Echo execution architecture.

Echo could be used in multiple ways, for example: (1) as a storage instance for a single multithreaded application, (2) as a storage system accessed in shared memory by multiple trusting applications, (3) as a server-based store for multiple untrusting processes, or (4) inside the OS, e.g., storing the data and metadata for a durable SCM-based kernel file system.

2.1 Architectural Model

Echo is a persistent, versioned, key-value store whose permanent data is maintained in non-volatile SCM. Figure 1 shows its basic execution model. On the left is a collection of *worker threads*, each with its own local store. The workers and local stores provide isolation and scalability; each worker thread can perform independent, thread-local storage operations without interference from the others. To write a consistent version of its local data to Echo's persistent store, a worker thread executes a commit transaction; this ensures that all modified data in the local store is written atomically and can be recovered if failure occurs during the process.

On the right of Figure 1 are one or more *master threads*, which perform most of the commit work in the master store. The figure shows a commit queue of requests from worker threads that the master threads service, processing one commit at a time and merging it into the master store.

As noted above, since DRAM and SCM differ in both performance and durability, we expect that SCM-based systems will contain both DRAM and SCM components. Our architecture exploits this dual-memory structure to benefit from the best properties of each memory technology. For this reason, Echo worker threads maintain their data in DRAM: that data is volatile, and access to it is fast. The master store is kept in SCM: the master is durable, but access (write access in particular) is slightly slower than to DRAM. This two-level scheme reduces writes to SCM, which also has advantages for write endurance.

2.2 Consistency Model

Key to Echo's architecture is a consistency model based on *snapshot isolation* [4]. Snapshot isolation trades full serializability for increased concurrency and performance because readers do not block writers. This tradeoff, used in a number

Operation	Description	Return Value
master_create()	Create new master store	store pointer
local_create(master)	Create new local store attached to master	store pointer
get(key)	Get key's value for worker's snapshot	value
put(key, value)	Put key-value in the local store	
delete(key)	Delete key from local store	
commit()	Commit local changes to master store	snapshot num.
set_current_snapshot(s)	Set local store to fetch from snapshot s	
get_version_iterator(key)	Iterate over all values for a key	version iter.
get_snapshot_iterator(s)	Iterate over key-value pairs in snapshot s	snapshot iter.
keep_snapshot(s)	Prevent collection of snapshot s	
delete_snapshot(s)	Mark snapshot s for collection	
delete_store(store)	Delete a master or local store	

Table 1: The Echo API. All operations except `create` take a pointer to a local store as a parameter.

of commercial systems where serializability is not critical, is available or standard in most database systems, such as Microsoft SQL Server, Oracle, and PostgreSQL [28, 34, 37].

Echo's database is a key-value store organized as a sequence of *snapshots*. A snapshot consists of a unique, monotonically increasing *snapshot number* and a set of key-value pairs. Threads manipulate the store using transactions; committing a transaction creates a new snapshot with a new snapshot number.

Each transaction thus executes under a thread-local current snapshot number, ensuring that all data it *reads* is consistent. However, thread *writes* to key-value pairs are made only in thread-local (volatile) memory. When a thread completes its transaction, it commits any modified data to the master store, creating a new snapshot that is durable and globally visible to all threads. This allows multiple threads to work independently and without coordination until a commit.

Snapshot isolation explicitly supports two of Echo's design goals. It supports scalability by allowing a high degree of concurrency on the part of worker threads. And it supports recovery because its transaction mechanism ensures data consistency following failure. Versioning is also inherent in this scheme: by keeping old data versions committed to SCM, a thread can explicitly back up to a previous snapshot and read a consistent set of data from the past.

2.3 API

Table 1 presents Echo's API. An application creates a persistent master store and multiple local stores (usually one per application thread) to connect to the master. A `put` operation copies a key-value pair to the local store. A `get` operation returns the key's value from the *master store* using the local store's *current snapshot number*.

A `commit` operation commits the key-value pairs in the local store's transaction to the master store. In addition, `commit` implicitly begins a new transaction, i.e., all values `put` to the local store between one `commit` and the next are written to the master at the next `commit`.

Echo includes a key `delete` operation, which is a `put` of a special "tombstone" value. Once the `delete` is committed to a snapshot S_i, all `gets` for that key from snapshot S_i or later will return *not-found*. If a value is `put` to the key in a later snapshot, S_j, then all `gets` after S_j will once again return a value. Because Echo retains historical data, `delete` does not immediately free any data. Instead, we rely on a storage management mechanism to free data while maintaining complete snapshots in the store.

2.4 Version and Storage Management

Versioning is an inherently space-expensive operation. As a consequence, Echo's architecture supports three alternatives for storage management. First, some applications may wish to keep all versions of the data they've committed. Second, an application can manually and precisely manage its storage; it does this by using the Echo API to explicitly delete specific snapshots no longer of interest, retaining only a subset of versioned data. Third, an application can choose Echo's *automatic version collector*, which uses a system-wide policy to reduce storage consumption. We do not discuss the details of storage management in this paper.

2.5 Failure Model

On a power-loss failure, the data committed to SCM at the time of the failure can be accessed on resumption and will be in a consistent state. All other application data, including stacks and uncommitted local storage, is inaccessible or potentially corrupted after failure. On power resumption, a root pointer to a master Echo store kept in SCM is provided to the application, which allows the application to initiate the recovery process. Hardware support for flushing data in the processors' caches (and possibly their CPU state, as well) to SCM on an impending failure may let us relax this failure model.

2.6 Architectural Implications of SCM

The use of PCM or other SCM technologies introduces several challenges that influence Echo's architecture. First is *write-performance*. While SCM reads are only slightly longer than DRAM's, SCM write latencies are expected to be many times greater than writes to standard DRAM. Therefore, Echo uses a hybrid, two-tiered approach, as previously described, in which threads maintain and manipulate local, volatile data in DRAM. We write only committed, persistent state to SCM, which avoids storing the metadata of every local `put` in SCM. This approach reduces accesses to SCM, making the system less dependent on write characteristics (such as latency and endurance), while letting threads perform thread-local work with the greatest possible performance.

The second challenge that influences Echo's architecture concerns *CPU caching*. SCM promises systems that are automatically persistent across power failures, but the limitations of today's processor architectures make this difficult to achieve in practice. Today's CPUs have large caches; when a power interruption occurs, dirty data that has not been evicted from the cache is lost, making recovery difficult. Recent work on SCM storage solves this problem by flushing data after it has been stored in memory [10, 43, 44]. This approach requires care for correctness and may hurt performance; e.g., flushed cache lines on the x86 are written back and invalidated, which causes a miss on the next read. Flushing thus causes more reads and writes to memory on top of the added write cost to SCM. Our separation of committed state from temporary state ensures that the cost of flushing is paid only for committed state.

SCM's third technical challenge is the *reachability problem*, which causes issues with common memory allocation mechanisms. Use of the standard C `malloc` in an SCM system can cause the application and the library to have different views of memory following failure. While this behavior is difficult to resolve, Echo is built so that it can be recovered to a consistent state after a failure.

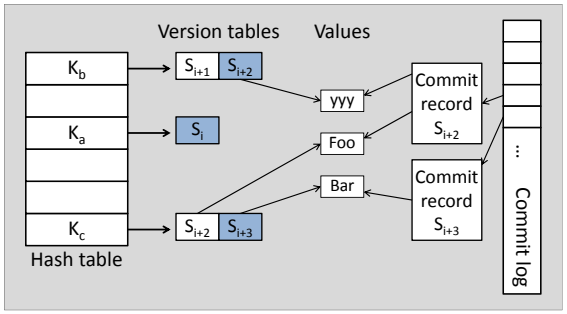

Figure 2: Simplified structure of the master store. Only a subset of the links and data is shown.

3. IMPLEMENTATION

We now focus on a brief discussion of the data structures and low-level details that provide insight into Echo's implementation. The Echo prototype is a shared library implemented in approximately 8,000 lines of C code. The library is included by applications and accessed through the API shown in Table 1. The majority of Echo's code resides in an *access method* module that implements `puts` and `gets` to storage system data structures.

For memory management, we use the `tcmalloc` library [18] for its scalability and performance benefits relative to the standard Linux glibc `malloc` library. We access `tcmalloc` through a set of Echo-specific wrappers, as described below. Threading and synchronization rely on the standard Linux `pthread` library.

3.1 Echo Data Structures

Echo supports a key-value storage system with multiple volatile local stores and a single persistent master store. Our prototype implements key-value storage using hash tables. While other data structures (such as trees of various types) might be suitable as well, we chose a hash table for its low-latency access at the cost of marginally greater storage space. Future work may incorporate tree structures as a means of supporting range queries, if needed. Both local and master stores use the same underlying data structure. The code for stores is unified, with minor differences; each hash table knows whether it is a local or master store and performs or omits certain steps based on this knowledge.

Figure 2 shows the master store's organization. Its four main data structures, all stored in SCM, include:

1. A *hash table* used to lookup version tables in the store
2. A per-key *version table*, consisting of a vector of *version table entries* that maintains the history of committed values for the key
3. A *commit log* that records and sequences commits registered by multiple worker threads
4. A set of *commit records*, each of which describes a single transaction and includes a list of the key-value pairs to be committed to the master store

Data values are allocated independently in the SCM heap and are pointed to by commit records and version table entries. The commit log contains pointers to the commit records in the order they were committed.

A version table entry (VTE) contains a pointer to the data in SCM, its size, snapshot number, and a pointer to the commit record for this key-value pair. Commit records facilitate conflict detection and recovery. A commit record describes a transaction and contains a list of key-value pairs to be committed (or already committed if the transaction completed successfully). It also contains beginning and ending snapshot numbers, used to determine the range of conflicts. The ending snapshot number is a unique identifier for the committed transaction.

A local store is a simplified version of the master; it keeps only a single value and a unique VTE per key rather than a version table. Structures in the local store are allocated from the volatile heap in DRAM. Each transaction starts with an empty local store. For each `put` request, the worker thread creates or modifies a VTE in its local store. A `get` request returns the value from its local store if one has been `put` during the current transaction. If the local store does not contain the requested key-value pair, the thread fetches the value from the master store using the current snapshot number.

Note that the structure and organization of Echo's master store relies on its ability to perform direct, fine-grained, byte-addressable reads and writes to durable SCM storage. Echo can simply and cheaply allocate (and deallocate) variable-sized blocks of durable heap memory for its persistent store. Recoverability is ensured through transactions committed directly into this memory through CPU instructions. In contrast, existing high-performance storage systems (key-value or otherwise) must use block-oriented optimization techniques – such as write-ahead logging, group commits, or clustering of data – in order to achieve I/O efficiency and durability on SSDs or disks.

3.2 Committing Transactions

An application thread performs a `commit` to make the data in its local store persistent and visible to other workers in the system. In Section 2.1, we described the Echo architecture conceptually using a collection of worker threads and a separate pool of master threads; however, in our current implementation, local worker threads assume the role of a master thread on a `commit` and immediately merge it into the master store.

To begin the commit process, the local worker allocates a commit record in SCM. It then iterates through all key-value pairs in its local store, copying each value into SCM and inserting into the commit record the corresponding key and its value pointer. Upon completion, it appends a pointer to the record to the commit log, at which point the transaction is formally committed and completely recoverable. The worker thread then assumes the role of a master *merge thread* and inserts the key-value pairs from the commit record into the master store while checking for conflicts. We leave the discussion of conflicts, aborts, and rollbacks to future work.

When the local worker performs its first `get` or `put` operation in a new transaction, its current snapshot number is automatically set to the greatest committed snapshot number in the master store. Snapshots beyond this are not made visible to local workers because they have either aborted or not yet committed.

3.3 Lightweight Versioning

Having described the basic transaction mechanism, we briefly note that the support of versions in an SCM-based store such as Echo is extremely lightweight. Versioning makes sense for persistent memory because it leverages the techniques already used to provide transactional updates to persistent data. In order to provide fail-safe transactions, local workers "append" their new values into durable memory at the beginning of the commit. The system cannot overwrite existing values in-place, which would cause inconsistencies in the case of a failure. Therefore, since we cannot update values in-place, there is little cost to keeping previous versions around for the benefit of applications.

A worker can get keys from older snapshots by changing its current snapshot number to a previous version using `set_current_snapshot`. In general, access to historical data is read-only; while the worker may still perform puts, any attempt to commit with an old snapshot number is likely to abort due to conflicts.

3.4 Hardware Implications

Our implementation makes several assumptions about the underlying hardware architecture. We assume cache flushing capability, which is needed to guarantee durable writes to SCM from current processors. Echo also requires some control over instruction ordering. These two instructions are available in all modern x86 processors [20]. We also assume that dirty cache lines evicted from the cache are written back to SCM in the presence of power failures; the amount of power required to ensure this in PCM DIMMs is discussed in [11].

We previously described several SCM-related issues, such as the impact of hardware data caching and the need for flushing. To address these problems consistently, we developed a set of functions that wrap the standard C memory-allocation functions in `tcmalloc`'s library. These wrappers implement the additional necessary steps to ensure correctness and durability.

Lastly, when allocating and updating data structures residing in the non-volatile heap, we always follow a series of steps to ensure that the state of a data structure. The last field in every data structure is its *state*: when a data structure is initialized or updated, the other fields of the structure must be updated and flushed to memory first; only afterwards is the state field updated to a new value and flushed. This two-step process ensures that the state field is never made persistent in SCM before the rest of the data structure has been safely stored, thereby ensuring consistency on recovery after failure.

4. EVALUATION

This section provides a simple measurement of Echo's performance, including latency, throughput, and scalability for various workloads.

We compare Echo's performance to two key-value stores that represent optimized approaches at both ends of the spectrum of possible SCM adoption. First, we compare to Google's LevelDB [19], a high-performance key-value store designed for conventional block-storage devices. Second, we compare Echo to Masstree [26], an in-memory key-value store. For a fair comparison, we: (1) placed Masstree's log in a ramdisk to approximate porting it to SCM, as we did with LevelDB, and (2) copied values returned by `get` operations in Masstree to a separate buffer to match the semantics of Echo and LevelDB (by default, Masstree just returns a pointer).

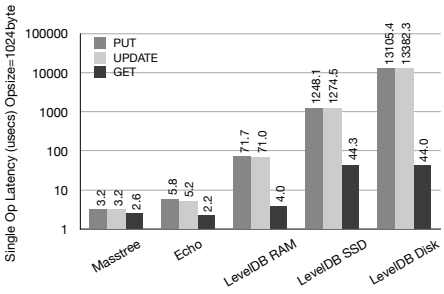

Figure 3: Average latency per operation for value sizes of 1K bytes.

LevelDB, Masstree and Echo are all distinct systems with different sets of features, making their direct comparison challenging. More specifically, there are two important feature differences to note. First, LevelDB and Masstree support ordered keys (and hence range queries over keys), while Echo does not, since it uses a hash table as its core data structure. This is likely to account for some of the performance differences between the systems, but it is important to note that Echo's architecture is not tied to a hash table as the core data structure — using a hash table is just an initial design decision for us. Second, Echo supports continuous data versioning, something that the other systems do not support.

We, like previous studies, [10, 11, 32, 43, 44] assume that hardware support will be provided for functionality such as cache flush on power failure. Because we assume that this will be default behavior, we do not explicitly flush the cache for any of the stores. Further, our evaluations approximate the behavior of SCM using DRAM, which allows us to run experiments on native hardware. This is appropriate due to cache hit rates in the absence of flushing, as well as the read-heavy workloads we evaluate on (assuming that read latency is comparable to DRAM read latency). However, both cache miss rate and memory access latency have the potential to affect Echo's performance, given widely varying workloads.

We ran all experiments on an AMD Magny-Cours Opteron with quad 1.9 GHz 12-core processors (48 cores total) and 64GB of memory, running Red Hat Enterprise Linux Server release 6.3 (Santiago). The SSD disk was a 180GB high performance 520 series Intel Solid State Drive with a SATA 3.0Gb/s connection. The disk and ramdisk sizes were 120GB and 4GB, respectively. We used ext4 file systems.

4.1 Basic Performance: Single-Operation Latency

As a baseline, we examine the latency of a single `get` or `put` operation in a single thread over the three systems. Our experiments used a fixed value size of 1K bytes, with measurements averaged over several thousand operations to remove anomalies. We ensured that `puts` were *durable* by performing a commit (Echo) or sync (LevelDB) after every `put`. There is no commit/sync operation after Masstree `puts`, since Masstree does not support such operations.

Figure 3 shows the latency (log scale) for single `get` or `put` operations on the various systems. Echo's `get` latency is about

the same as Masstree's.[1] Echo's `put` latency is about 2x slower than Masstree.

The figure also shows LevelDB with ramdisk, SSD, and moving-head disk. Compared to LevelDB operating in DRAM (LevelDB-RAM), Echo is approximately 1.79x faster for `gets` (2.2 usec vs. 4 usec), 12.25x faster for `puts` to new keys (5.9 usec vs. 71.7 usec), and 13.73x faster for updates (5.2 usec vs. 71.0 usec). The comparatively low `get` time for the three LevelDB configurations shows the effectiveness of LevelDB's memory caching. Not surprisingly, durable `puts` on SSD and disk are orders of magnitude slower than in DRAM ramdisk. We omit these configurations for clarity in the second diagram.

4.2 Scalability: Throughput Scaling with Core Count

Increased concurrency can lead to increased synchronization contention on metadata accesses, potentially reducing scalability. To examine this impact, we ran a synthetic workload experiment to measure aggregate operation throughput over all cores to show how the various systems scale with core count. An experiment launches n threads, each pinned to a separate core for fairness. Each thread performed random accesses on the store for a fixed period of time (about 5 seconds), gathering statistics on the number of `puts` and `gets` performed. Note that this is a *worst-case* scenario since all threads are performing continuous store operations in a tight loop.

Figure 4 shows the throughput of a mixed workload of 80% `gets` and 20% `puts` for Echo, LevelDB, and Masstree, as a function of the core count. We see that even when run on ramdisk, LevelDB's scalability is seriously limited and far below the other systems.

For this experiment we measured two versions of Masstree: one with logging and checkpointing for persistence, and one with those features turned off (i.e., a memory-only, nondurable Masstree). The throughput of memory-only Masstree exceeds that of Echo by a small factor, which increases with core count. Though it provides durability, the throughput of the logging Masstree configuration is below that of Echo due to the cost of background threads accessing structures and incurring filesystem overhead for durability. Therefore, we see that persistence is not free, even for a key-value store that is highly optimized for memory.

5. RELATED WORK

Non-volatile Memory Systems. Previous research efforts have examined the use of SCM for different abstractions in persistent memory. Mnemosyne [44] supports *persistent regions* that user code can map into its virtual address space and manipulate directly through word-level operations. NV-Heaps [10] supports a *persistent object heap*, focusing on safe pointers and programming. Both provide transaction mechanisms for concurrency and failure recovery. BPFS [11] is a *file system* that maintains a persistent in-memory tree for file data, directories, and i-nodes; its copy-on-write ("short-circuit shadow paging") mechanism uses byte addressability to perform fine-grained file updates in SCM. Echo operates at a higher level than Mnemosyne and NV-Heaps, providing a persistent key-value store with a standard get/put interface for applications; it benefits from snapshot isolation for concurrency, consistency, and versioning and leverages DRAM along with SCM to increase performance and SCM write lifetime.

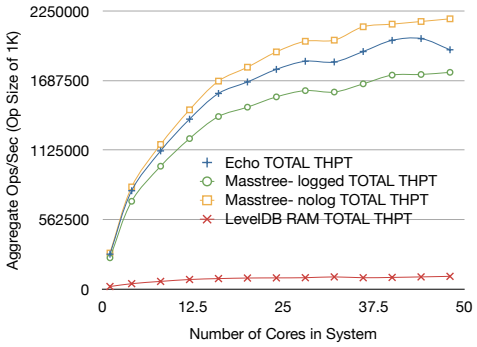

Figure 4: Average total throughput as the number of threads increases, with a workload of 80% `gets`.

Venkataraman et al. [43] present consistent and durable data structures (CDDSs) using a B-tree that allows in-memory updates on a single copy. Rio Vista [25] combines a battery-backed DRAM file cache with a recoverable memory library to improve transaction performance for a disk-based system. Narayanan and Hodson [32] provide a study of "whole-system persistence" for legacy applications, which operates by flushing state to SCM on failure, arguing that the flush-on-fail approach is superior to in-memory persistent heaps.

Several projects have considered hybrid memory systems, particularly using NAND flash (SSD). Mogul et al. [29] describe systems using both DRAM and flash, with OS support for optimizing migration between the two. SkimpyStash [12] is a key-value store tailored for flash with a highly optimized DRAM lookup layer, the contributions of which are largely orthogonal to Echo. eNVy [45] uses a flash-based storage system with a small battery-backed SRAM buffer to hide latency, and Qureshi et al. [38] describe the advantages of a PCM-based main memory system with a small DRAM buffer. Like BPFS and Mnemosyne, Echo assumes a memory architecture consisting of both DRAM and NVRAM accessible on the memory bus. To our knowledge, Echo is the only long-term, key-value storage system designed for SCM persistent memory.

Versioned Systems. Many versioned systems have been studied in the past. For example, Driscoll et al. [15] developed techniques for maintaining all versions of linked data structures. The KeyKOS OS implemented efficient full-system checkpointing to disk [22]. Many file systems provide versioning or checkpointing: Elephant [41] automatically retains all versions of user files; ZFS [35] uses copy-on-write to enable faster file system snapshots, and the recent Btrfs [7] supports both readable and writable snapshots, as well. Echo integrates a versioned hash table with its write-ahead commit log to achieve versioning at little cost on top of the mechanisms already used for transactional updates.

[1] As noted above, we modified Masstree to copy the 1KB `get` data into a buffer to match Echo's and LevelDB's behavior. Just reading the `get` data from Masstree through a returned pointer would save approximately 20% from the time shown, and returning the pointer only (the default) would be 2x faster than the read-only case (around 1 usec).

Transactions and Snapshot Isolation. *Multiversion concurrency control* (MVCC) applies timestamps or version numbers to data, allowing each client to read data consistent with its current timestamp [5]. The main benefit of MVCC over other database techniques, such as two-phase locking, is that readers do not block writers of the same data; this enables increased transaction throughput if conflicts between writers are infrequent. Many versioned data systems are based on MVCC, often implemented using copy-on-write (e.g., CDDS and BPFS). Echo implements snapshot isolation, which is a form of MVCC; however, while most MVCC systems discard old versions when no readers remain, Echo maintains previous versions and provides efficient historical access and garbage collection.

MVCC is still the standard for high-performance transaction processing systems with strong isolation guarantees, including Oracle [34] and MySQL [31], and for research systems such as Hyder [6]. Google uses snapshot isolation in Percolator [36] to perform incremental updates to its web index.

Key-value Stores. We have used Echo as a shared library linked to application code; Berkeley DB [33] and LevelDB [19] are libraries offering persistent key-value storage, but they are designed for disks rather than SCM. FlashStore [13] and BufferHash [1] are key-value stores optimized for SSDs. Masstree [26] is a highly optimized, key-value store that achieves high throughput across many cores by using multiple tree structures and optimistic concurrency control in memory; it uses write-ahead logging and periodic checkpoints to disk for durability. Echo maintains its versioned, durable store directly in SCM, using transactions to commit multiple values at once for consistency on failure.

Key-value stores have also become extremely prevalent as infrastructure backing the Web [2, 9, 14, 24, 40]. Distributed key-value stores are attractive because they can be scaled-out much more aggressively than traditional relational databases. These systems offer a wide range of data models, consistency and availability guarantees, and support for multi-key updates. For Echo, our focus is on efficient versioning and the use of non-volatile memory rather than on these properties. However, we believe that Echo would make a suitable backing store for many of these distributed key-value stores on future servers with SCM.

6. CONCLUSIONS

This paper presented the design and implementation of Echo, a key-value storage system explicitly designed for future storage-class memory technologies. Echo is intended to provide the durability guarantees of block-storage systems while achieving the performance and scalability commensurate with state-of-the-art key-value stores. Key to Echo's design is its two-level memory structure and the use of snapshot isolation for concurrency, consistency, and versioning. The simple measurements presented show the potentials of Echo's esign to meet our goals.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. This work was supported by NSF grants CNS-1217597 and CNS-1016477, NSF Graduate Research Fellowship DGE-0718124, the Wissner-Slivka Chair, and gifts from Intel Corporation and Google.

7. REFERENCES

[1] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *NSDI*, pages 433–448, 2010.

[2] Apache. Apache Cassandra. `http://cassandra.apache.org/`.

[3] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating System Implications of Fast, Cheap, Non-Volatile Memory. In *HotOS*, 2011.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[5] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[6] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *CIDR*, pages 9–20, 2011.

[7] Btrfs. `https://btrfs.wiki.kernel.org/`.

[8] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion Recovery for Database-backed Web Applications. *SOSP*, 2011.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI*, 2006.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. *SOSP*, 2009.

[12] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. *SIGMOD*, 2011.

[13] B. K. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *PVLDB*, 3(2):1414–1425, 2010.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[15] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC*, 1986.

[16] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *OSDI*, 2002.

[17] I. Ekin and A. Goel. Data Recovery for Web Applications. *Dependable Systems and Networks (DSN)*, 2002.

[18] Google. Fast, multi-threaded malloc and nifty performance analysis tools. `http://code.google.com/p/gperftools/`.

[19] Google. Google LevelDB. `http://code.google.com/p/leveldb/`.

[20] Intel. Intel 64 and IA-32 architectures software developer's manual. http://download.intel.com/products/processor/manual/325462.pdf, July 2012.

[21] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion Recovery Using Selective Re-execution. *OSDI*, 2010.

[22] C. R. Landau. The checkpoint mechanism in KeyKOS. *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, September 1992.

[23] B. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1), 2010.

[24] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *SOSP*, 2011.

[25] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *SOSP*, 1997.

[26] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[27] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for JavaScript Applications. *NSDI*, 2010.

[28] Microsoft. Microsoft SQL server: Working with snapshot isolation. http://msdn.microsoft.com/en-us/library/ms130975.aspx.

[29] J. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for NVM + DRAM hybrid main memory. *HotOS*, 2009.

[30] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer. Provenance for the Cloud. *FAST*, 2010.

[31] MySQL. InnoDB multi-versioning. http://dev.mysql.com/doc/refman/5.6/en/innodb-multi-versioning.html.

[32] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS*, 2012.

[33] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[34] Oracle. Oracle database concepts 11g release 2: Introduction to data concurrency and consistency. http://docs.oracle.com/cd/E11882_01/server.112/e25789/consist.htm#CNCPT121.

[35] Oracle. Working with Oracle Solaris ZFS snapshots. http://www.oracle.com/technetwork/server-storage/solaris11/documentation/zfssnapshots-365269.pdf.

[36] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[37] PostgreSQL. Concurrency control. http://www.postgresql.org/docs/9.2/static/mvcc-intro.html.

[38] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA*, 2009.

[39] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev.*, July 2008.

[40] Redis. http://redis.io/.

[41] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *SOSP*, 1999.

[42] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453:80–83, May 2008.

[43] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, 2011.

[44] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.

[45] M. Wu and W. Zwaenepoel. eNVy: a non-volatile, main memory storage system. In *ASPLOS*, 1994.