Hathi: Durable Transactions for Memory using Flash

Mohit Saxena
U. Wisconsin-Madison
msaxena@cs.wisc.edu

Mehul A. Shah, Stavros Harizopoulos
Nou Data
{mashah,stavros}@gmail.com

Michael M. Swift
U. Wisconsin-Madison
swift@cs.wisc.edu

Arif Merchant
Google
aamerchant@google.com

ABSTRACT

Recent architectural trends — cheap, fast solid-state storage, inexpensive DRAM, and multi-core CPUs — provide an opportunity to rethink the interface between applications and persistent storage. To leverage these advances, we propose a new system architecture called Hathi that provides an in-memory transactional heap made persistent using high-speed flash drives. With Hathi, programmers can make consistent concurrent updates to in-memory data structures that survive system failures.

Hathi focuses on three major design goals: ACID semantics, a simple programming interface, and fine-grained programmer control. Hathi relies on software transactional memory to provide a simple concurrent interface to in-memory data structures, and extends it with persistent logs and checkpoints to add durability.

To reduce the cost of durability, Hathi uses two main techniques. First, it provides *split-phase* and *partitioned commit* interfaces, that allow programmers to overlap commit I/O with computation and to avoid unnecessary synchronization. Second, it uses partitioned logging, which reduces contention on in-memory log buffers and exploits internal SSD parallelism. We find that our implementation of Hathi can achieve 1.25 million txns/s with a single SSD.

1. INTRODUCTION

Transactions serve two purposes that make it easier to build robust applications. First, transactions enable fine-grained concurrency control, allowing developers to scale applications more easily across multiple processors [8, 20]. Second, transactions provide a simple interface for managing the durability and consistency of application state in the face of failures [16].

However, the common interfaces to ACID transactions – DBMSs or distributed transaction systems – are a poor fit for many modern workloads, partially due to the cost and complexity of managing such systems. As a result, many applications, particularly web services, sacrifice strong consistency for simpler storage models, such as key-value stores [14]. We believe that a key challenge of existing transaction interfaces is that they require the use of large, complex systems (the DBMS or distributed transaction coordinator) even for lightweight operations. In contrast, most file systems reject transactional semantics and provide only coarse-grained control over durability in the form of flushing pages or files to disk.

In this paper, we describe a lightweight, high-performance and ACID-compliant transactional store based on *durable memory transactions*: programs can concurrently update in-memory data structures that are consistent and persistent across failures. We leverage three recent architectural trends in the design. First, DRAM prices have dropped to a point that even mid-tier servers support up to 4 TB of memory. At these sizes, many workloads can execute in core rather than from disk — an observation also made by others [19, 23]. Second, power considerations have driven processor manufacturers away from uni-processors towards multi-core chips, so concurrency between threads becomes a key concern. Third, flash-based solid-state drives (SSDs) provide scalable bandwidth and 1-2 orders of magnitude lower latency than the fastest disks.

Hathi is a high-speed, durable, main-memory transactional store that leverages these trends. It presents an in-memory transactional heap interface that is automatically made persistent on fast SSDs. Thus, programs can create and manipulate in-memory data structures, but ensure the data is durable with little extra effort. To do so, Hathi combines the simple and highly concurrent interface ("ACI") of transactional memory [20] with an SSD-optimized write-ahead logging and checkpointing scheme for durability ("D"). Thus, a programmer can wrap a section of program in a transaction to make updates durable and consistent. As flash storage is fast but still much slower than memory, Hathi implements two approaches to reduce and eliminate the overhead of persistence.

First, Hathi provides options at transaction commit to control whether the program blocks, similar to but more fine-grained than asynchronous file I/O. This control has not previously been available for memory transactions, and allows developers to leverage application knowledge for increased performance. Specifically, Hathi exposes *split-phase* commit, which decouples the installation of in-memory updates from the flush of transaction log records. Using this interface, applications can continue with other tasks and later check for completion of the commit, thereby overlapping computation and commit I/O.

Second, Hathi uses partitioned logging, in which each thread maintains a separate log. Partitioned logging leverages the SSD's internal parallelism and avoids contention on in-memory log buffers that hold the tail of the transaction log. For full consistency, Hathi ensures that all preceding transactions from all threads are durable before marking a transaction as durable. However, Hathi also provides *partitioned commit*, which allows application threads to commit transactions that operate on independent data structures without coordinating with the logs from other threads. With these interfaces, applications retain the recoverability guarantees of synchronous commit at speeds closer to asynchronous commit. Our

Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN 2012), May 21, 2012, Scottsdale, AZ, USA.

experiments with these interfaces show 4-5x throughput improvements over synchronous commit.

With these optimizations, we show that Hathi reaches 1.25 million txns/sec on a high-end FusionIO drive and nearly 200 K txns/sec on a consumer-grade Intel X-25M SSD. Thus, Hathi provides durable transactions at little additional cost over non-durable transactional memory. At these speeds, we believe Hathi is suitable for building not only user-facing applications, but also infrastructure applications like file systems, key-value stores, massively multiplayer online games and social-network graphs.

Persistent memory [24, 29] and persistent object stores [27, 30] have been proposed in the past. However, Hathi is the first to rely on commodity processor and storage technology (i.e., no phase-change memory or battery-backed DRAM), and operate at near-DRAM speed, courtesy of its interfaces for split-phase and partitioned commit. In addition, Hathi provides a low-level unstructured memory space that makes few demands of the programmer, allowing it to be used as a building block for these systems. In summary, Hathi provides a simple application interface, fast and scalable mechanisms for transaction commit, and fine-grained durable memory transactions.

2. BACKGROUND

Hathi is enabled by recent developments in solid-state drives (SSDs). Internally, SSDs are comprised of multiple flash chips accessed in parallel. As a result, they provide scalable bandwidths limited mostly by the interface between the drive and the host machine (generally SATA or PCIe) [7, 11]. In addition, SSDs provide access latencies orders of magnitude faster than traditional disks. For example, FusionIO enterprise ioDrives provide 25 μs read latencies and up to 600 MB/sec throughput [1]. Consumer grade SSDs provide latencies down to 50-85 μs and bandwidths up to 250 MB/sec [18]. To fully saturate the internal bandwidth of the device, enterprise SSDs support longer I/O request queues.

With current technology, SSD bandwidth is comparable to or exceeds network bandwidth (1.9 Gb/s read, 0.7 Gb/s write for a commodity SSD [18]). However, the performance of random writes may be much lower, consumer-grade devices can only provide 3000-8000 random writes per second. Thus, sustaining fine-grained updates at full network latency requires mechanisms to tolerate the write latency and queueing delays from prior writes.

3. INTERFACE AND DURABILITY OPTIONS

Hathi provides programmers with a familiar set of simple primitives that facilitates them to build fast, robust, and flexible persistent memory regions. Rather than forcing programs to use low-level file primitives or convert their data into a database format, Hathi enables a program to use any in-memory data structure for durable data with *persistent heaps*. Heaps are persistent memory regions that applications can read or write using a software transactional memory (STM)-like interface. Hathi provides a `pmalloc` interface to create a heap, which allocates a segment of memory and associates it with a checkpoint file on an SSD. A program can then perform consistent reads and writes to the heap using the interface shown in Figure 1. A read from a given heap address and size copies the memory region to a user-specified buffer, and a write to a heap updates an in-memory copy of the data. A transaction can abort, which erases all updates, or commit, which makes updates persistent and visible to other threads.

Persistent memory is an invaluable asset for applications that require both high throughput and durability. Some such applications include file system journaling, high-throughput main memory key-value stores, social network graph databases, main memory transaction processing systems, persistent logs for highly available network servers, and massively multiplayer online games [10, 28].

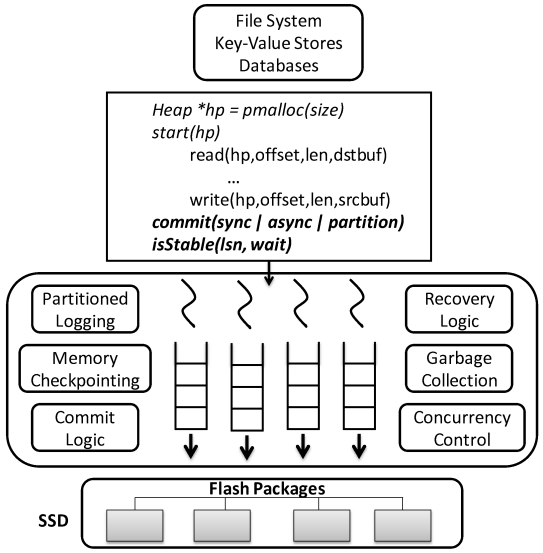


Figure 1: Hathi architecture and interface.

```
hash_insert(hashtable, key, value) {
    txn_start(hashtable);
    if( txn_read(hashtable->entries)
            > txn_read(hashtable->loadlimit) ) {
        hashtable_expand(hashtable);
    }
    bucket = pmalloc(sizeof(*bucket));
    hash = getHash(hashtable, key);
    index = indexFor(hashtable->length, hash);
    txn_write(bucket->key, key);
    txn_write(bucket->value, value);
    txn_write(bucket->next, hashtable->table[index]);
    txn_write(hashtable->table[index], bucket);
    txn_commit(sync);
}
```

Figure 2: Hathi hash table insert: example usage of memory transactions.

For example, Figure 2 demonstrates how to update a hash table within a transaction. The transaction encompasses both data reads, to provide concurrency control, memory allocation, and updates. At transaction commit, a runtime system ensures the updates are consistent (i.e., no interleaving with other transactions) and durable. As the transaction executes, Hathi writes the updates to a per-thread log, and then forces the log to an SSD to make the update durable.

File systems and databases provide disk-based journaling capabilities for transactional updates. Hathi provides durability for main

DEPENDENTTRANSACTIONS

START(hp)	START(hp)
READ(hp,0,10,buf)	READ(hp,11,10,buf)
for(alli)	for(alli)
buf[i]+=1	buf[i]=2
WRITE(hp,11,10,buf)	WRITE(hp,0,10,buf)
COMMIT **sync**	COMMIT **sync**

PARALLELTRANSACTIONS

START(hp)	START(hp)
READ(hp,0,10,buf)	READ(hp,11,10,buf)
for(alli)	for(alli)
buf[i]+=1	buf[i]=2
WRITE(hp,0,10,buf)	WRITE(hp,11,10,buf)
COMMIT **partition**	COMMIT **partition**

Figure 3: Dependent and parallel transactions.

memory transactions through a log stored on an SSD. Although flash latencies are low, they are much larger than latencies to main memory. So, programmers still must be careful about when to wait for updates to become durable. Hathi provides programmers with three options to commit that control its durability guarantee: *sync*, *async*, and *partition*.

Sync and Async. *Sync* and *async* are similar to database synchronous and asynchronous commit and the `fsync()` file system operation. Synchronous commit only returns after forcing the transaction's log records *and* the records of all preceding transactions' to stable storage. This ensures that the update is durable and application data structures are consistent, because all prior updates were also written to the SSD. In contrast, asynchronous commit returns as soon as the transaction finishes updating memory, and does not wait to force the log records to storage. Even with this option, the heap recovers to a consistent point in the transaction history, although it may lose recently committed asynchronous transactions.

Partition. Commit's third option, *partition*, relaxes the isolation guarantee for better performance. Hathi has a separate log partition for each thread (see Section 4). Partition commit simply forces the log for the local thread. This option is useful when an application uses transactions for atomicity and durability but *not* for isolation, and may be used when each application thread operates on different (partitioned) data. Such partitioning may be easy when updating regular data structures such as a hashtable or a matrix.

This commit option potentially allows for more overlap between computation and I/O across threads than synchronous commit, because a thread need not wait for preceding transactions in other threads to become stable. Although applications can mix this option with the others above, they must be careful to ensure the independence of partition commit. Figure 3 shows an example of dependent transactions, which cannot use partition, and parallel ones that can.

isStable. Hathi provides an additional interface to query whether a prior asynchronous transaction is durable. On success, async commit returns the logical sequence number (LSN) for the transaction. The *isStable(lsn,wait)* call indicates whether a transaction with that LSN is stable and recoverable, and optionally waits until it is. A transaction is recoverable if all transactions it is dependent on are durable, which may be all preceding transactions. This interface allows applications to make commit *split-phase*: initiate commit early, and then wait for it to complete later. For example, an event-driven server can continue with other client transactions and return results once the log flushes. In this way, Hathi can overlap I/O and computation, getting recoverability at nearly the same throughput as asynchronous commit.

4. DESIGN AND IMPLEMENTATION

Unlike traditional databases, Hathi does not maintain a backing store: storage contains only logs and checkpoints, and the logs contain only redo records of updates. Checkpoints are copies of the heap kept on an SSD, and allow trimming the logs to reduce recovery time. Hathi checkpoints the heap incrementally to avoid stalling the system.

We implemented a prototype of Hathi by modifying an existing software transactional memory system (TinySTM [15]), used for concurrency control, to add write-ahead logging and recovery for durability. Hathi's transaction API wraps the underlying STM calls, which maintain a log of updates to apply when a transaction completes. The STM ensures transactions are isolated by acquiring locks on each memory location referenced, and will abort one or more transactions when it detects conflicting lock request. On successful commit, Hathi tags the transaction with a logical sequence number (LSN) given by the STM and inserts them into the thread's log. The LSN is a global counter that the STM atomically increments before releasing all locks, thus ordering all transactions. For all commit types, Hathi reflects the transaction updates in-memory and releases locks before the log records reach the SSD to allow other threads to proceed.

Partitioned Logging. Hathi employs *partitioned logging* both in memory and in storage: each core maintains its own transaction log that can be independently flushed to a separate location in storage. Merging the logs provides a logical global log. Partitioned logging is well suited to both SSDs and multi-core architectures: SSDs require multiple outstanding requests to saturate their bandwidth, and partitioned logging allows multiple cores to generate requests simultaneously. In addition, partitioned logging reduces lock contention, since threads access their local log without synchronization. Hathi further reduces latency with direct I/O to bypass the file-system buffer pool.

Partitioned logging complicates recovery by raising the possibility that later transactions from one thread will become durable before earlier transactions of another. This potentially leaves a gap in the transaction sequence on failure, resulting in an inconsistent application data structure. During recovery, it may therefore be inconsistent to replay all committed transactions in all logs. On recovery, Hathi takes care to only replay transactions up to the first missing transaction.

Thus, Hathi maintains two invariants that tie together logging and checkpointing for correctness of its recovery algorithm. First, each transaction has an LSN that is consistent with the partial order of transaction dependencies; a transaction can only depend on transactions with lower LSNs. When logging, a transaction is not recoverable until all preceding transactions in this total order are recoverable. Second, to ensure that all updates are consistently applied to a checkpoint, we must maintain the *write-ahead logging discipline*: a chunk of memory in a checkpoint cannot be used for recovery until the effects of all transactions reflected in that chunk have been made recoverable.

The Hathi interface enables applications to control durability of their data. Hathi maintains a global variable, `min_lsn`, that tracks the youngest recoverable transaction. Each transaction log maintains the latest LSN that is on the non-volatile store; the `min_lsn` is the minimum or oldest of these. Each thread updates this variable after flushing its log. For synchronous commit, Hathi flushes the local log and waits until `min_lsn` exceeds or equals the transaction LSN. To improve throughput, synchronous commit batches multiple transactions into a single log flush, a technique called *group commit* [17].

Partitioned commit annotates the transaction's log record with a *partition* flag, flushes the log, and does not wait for `min_lsn`. The flag indicates that the transaction can safely be recovered, even if preceding transactions from other threads are not available. Asynchronous commit also does not wait and, like group commit, defers forcing the log until either a fixed time period elapses or a fixed

amount of log space is used. Thus, partitioned commit provides the durability semantics of synchronous commit for the local log, and the performance of asynchronous commit across logs. Finally, `isStable` compares the given LSN against `min_lsn` and waits if necessary.

Algorithm 1 Hathi Memory Checkpointing

```
1: for chunk in heap do
2:     tx_start
3:     tx_read(chunk,copyBuffer)
4:     chunkLSN = tx_commit(async)
5: end for
6: isStable(lastChunkLSN,true)
7: update checkpoint header
8: sleep(timer)
```

Checkpointing and Log Trimming. In checkpointing, Hathi periodically writes memory in configurable fixed-sized *chunks* to the SSD. When a checkpoint is needed, a separate checkpoint thread walks through the heap and writes out chunks, with each write protected by an STM transaction. This method ensures consistency with concurrent transactions, without the need to pause execution because both chunks and log records use the same log sequence number space. Although non-intrusive, this incremental checkpointing method allows for a transaction to straddle a chunk that has been checkpointed and one that has not. In such case, a chunk of memory in a checkpoint cannot be used for recovery until the effects of all transactions reflected in that chunk have been made recoverable, that is, written to disk so they can be re-applied to chunks that do not include their effects. Thus, a checkpoint is not valid until all transactions reflected in any of its chunks have been made recoverable (similar to write-ahead logging). Once Hathi has created a valid checkpoint of the entire heap, it discards unneeded older checkpoints and garbage collects log records prior to the earliest chunk in the new checkpoint since their effects are already included.

Algorithm 1 describes checkpointing. Since the workload fits in main memory, we can safely require that enough storage space is available for more than two checkpoints. Thus, we need not ensure that all transactions are durable before starting the checkpoint, as the previous checkpoint can still be used for recovery. Hathi first copies data from the persistent heap into a temporary buffer using an STM read, and then writes out the chunk and its LSN to checkpoint space. Once all chunks have been written, it writes a checkpoint header that indicates what checkpoints and log records can be garbage collected.

Recovery. Hathi performs recovery by loading a checkpoint and then replaying logs. It reads the checkpoint header to find the LSN of the oldest checkpoint chunk. Starting with that chunk, Hathi replays logs and checkpoints in LSN order until it reaches the end of one thread's log and a gap in the LSNs, which indicates a missing transaction. It then continues to scan logs and replays records labeled *partition*, which can still be safely applied.

In summary, Hathi provides novel interfaces and mechanisms designed to provide durability for memory-resident datasets and optimized for new flash SSDs and multi-core processors. Partitioned and split-phase commit interfaces provide low-level control over latency to application programmers. Partitioned logging and recovery, chunk-based incremental checkpointing and log trimming mechanisms optimize for the performance of flash SSDs which possess fast random access and high internal parallelism.

5. EVALUATION

Our current implementation of Hathi supports partitioned logging, incremental checkpointing, and recovery. In this section, we present experiments that show: (i) the cost of durability, (ii) the value of partitioned logging, and (iii) the performance tradeoffs of different durability options.

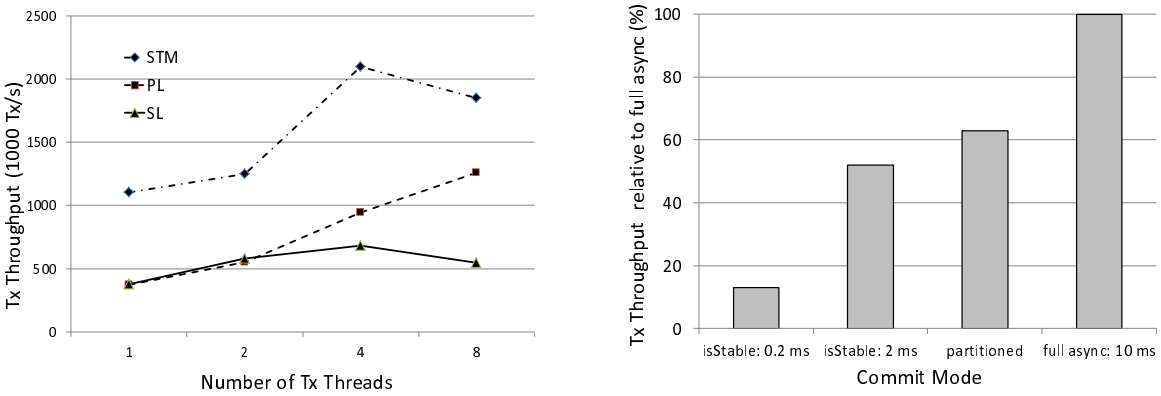
Methodology. We use two setups in these experiments. To emulate a high-end system, we use a 3.0 GHz Intel Xeon HP Proliant quad-core server with 8 GB DRAM and a 80 GB PCIe FusionIO ioDrive. On this, we run a synthetic workload of low-contention memory transactions for analyzing the overhead of providing durability. Each thread continuously executes transactions that read and write six random words in a 4 GB heap. The second system runs the travel reservation workload from STAMP transactional memory benchmark suite [22] that we ported to Hathi. This ran on a 2.5 GHz Intel Core 2 quad with 1 GB heap and a consumer-grade SSD, an Intel X-25M. We use a 10 ms group commit timer and maximum 512 KB log buffer for each thread.

Durability Costs. Figure 4(a) compares the performance of the high-end system running with Hathi with durable transactions using partitioned logging (PL) and asynchronous commit, against the base STM system without durability (STM). The STM system peaks in throughput at 4 threads with 100% CPU utilization. After 4 threads, the throughput drops because of increased contention over the STM's commit lock. With Hathi at 8 threads, we reach 1.25 M txns/sec with 70% CPU utilization, which is only 38% short of the peak STM throughput. Similarly, on the STAMP workload, Hathi reaches nearly 200 K txns/sec, only 15% short of the STM-only throughput. These results indicate that the added cost of durability is low for these workloads with Hathi. As a comparison, recent work on persistent memory using future projections of phase-change memory performance achieved only 1.3-1.6 M txns/sec with 4 cores [12, 29], not much better than Hathi in the high-end configuration.

Partitioned Logging. Figure 4(a) also compares the transaction throughput for single log (SL) and partitioned log (PL) on FusionIO. PL is more than 130% faster than SL, which utilizes less than 20% CPU. Thus, the single log is clearly the bottleneck. Single log performance degrades after 4 threads because of write serialization to ensure sequential I/O. In contrast, as we increase the number of threads and log partitions with PL, the throughput increases almost linearly. We note that partitioned logging benefits from more concurrent I/O requests and low access latencies of the storage device, as the seek costs in a disk would make partitioned logging more expensive.

We also investigate the impact of logging on the average transaction latency of the high-end system with asynchronous commit. With 4 threads on FusionIO, the average transaction latency for flash is only 14 ms. This is higher than the group-commit latency of 10 ms due to the queuing delay behind other log flushes. In comparison, using a single log raises latency to 25 ms, again demonstrating the value of partitioned logging in keeping latencies down.

Durability Tradeoffs. Using the STAMP workload, we investigate the performance impact of different durability options. Figure 4(b) shows the performance of split-phase and partitioned commit with different time intervals between calling `isStable` to wait for all previous transactions to become durable. As expected, when ensuring durability every 0.2 ms, transaction throughput is only 13% of asynchronous commit. However, allowing transactions to stay in memory for 2 ms achieves 50% of asynchronous performance. Using partitioned commit in addition to split-phase commit increases performance by 20% and is only 40% below asynchronous commit. These results demonstrate that the use of split-phase and partitioned



(a) Durability costs

(b) Commit mode performance

Figure 4: Durability costs and commit modes' performance.

commit to make a set of updates durable provide a middle-ground between the performance of asynchronous commit and the recoverability of synchronous commit.

6. RELATED WORK

Hathi derives inspiration from past work on storage-class memory (SCM), main-memory data stores, and persistent objects and databases.

NV-Heaps [12] and Mnemosyne [29], investigated the use of SCM and new processor primitives to provide support for persistent memory. Hathi achieves the same goal using existing hardware and commercially available flash memory technology. FlashVM and SSDAlloc [25, 9] are hybrid SSD/RAM memory managers optimized for the performance characteristics of SSDs. They do not support Hathi's transactional semantics to provide durability for arbitrary main-memory data structures. Hathi is similar to past work on durable memory transactions, such as eNVy [31], RVM [24] and Rio Vista [21]. Hathi differs in its architecture, providing word-level persistence rather than the page level of RVM, and persists data to flash rather than relying on a battery or specialized memory controller, as in Rio/Vista and eNVy (for uncommitted data).

Persistent object stores such as Texas [27], QuickStore [30], Grasshopper and Cricket [13, 26] provide a high-level structured object-interface to applications. They also provide safety properties, such as ensuring pointers in persistent data structures reference only persistent data. Hathi provides a low-level unstructured memory interface over which these systems can be layered to provide the same guarantees. In addition, Hathi operates at near-DRAM speeds using commodity hardware because of its low-level commit interface that allows fine-grained control on transaction latency. FusionIO's Auto Commit Memory [2] is similar to Hathi and provides atomic and durable data updates with memory semantics for programming. In addition, Hathi provides general-purpose transactions with support for flexible commit interfaces to persist data.

Closely related to Hathi are other main memory data stores. These fall into three categories: relational stores, *e.g.*, TimesTen and VoltDB [5, 6]; object stores, *e.g.* GemStone and RamCloud [3, 23]; and key-value stores, *e.g.* memcached [4]. The key-value stores tend to reject transactional semantics. The relational and object stores are tuned for high-throughput transactions, but focus on scaling across a cluster and provide durability through replication to other machines rather than to a local storage device, as Hathi does.

7. CONCLUSIONS

Programmers trained in the era of disks have learned that persisting data requires complex software and rich interfaces to overcome the long latency to storage. However, large memory sizes, multi-core processors, and high-speed flash storage enable a new generation of storage interfaces that reduce the gap between volatile and persistent data. Rather than maintaining two copies of data in different formats, durable memory stores enable a single data representation, optimized for in-memory access, that can also be recovered reliably when failure occurs.

In this paper, we describe Hathi, a high-speed, main-memory, durable transaction store that harnesses recent technology advances. We show the use of existing hardware and persistent memory available today, without the need to wait for next-generation non-volatile memory technologies. Hathi provides a powerful interface that eases application development, and still retains much of the performance of the underlying hardware. The flexible interface of Hathi also opens up new opportunities for application developers to explore the use of different persistent memory data structures and declarative programming styles.

Acknowledgments

This work is supported in part by National Science Foundation (NSF) grant CNS-0834473. Swift has a significant financial interest in Microsoft.

8. REFERENCES

[1] Fusion-IO PCI-e ioDrive. `www.fusionio.com/products/iodrive`.

[2] FusionIO Auto-Commit Memory. `http://www.fusionio.com/blog/auto-commit-memory\\-cutting-latency-by-eliminating-block-i/o`.

[3] GemStone Object Server. `www.gemstone.com/products/gemstone`.

[4] memcached: High-performance Main-Memory Key-Value Store. `www.memcached.org`.

[5] Oracle TimesTen In-Memory Database. www.oracle.com/timesten.

[6] VoltDB: SQL DBMS with ACID. www.voltdb.com.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX*, 2008.

[8] M. K. Aguilera, A. Merchant, M. A. Shah, A. C. Veitch, and C. T. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[9] A. Badam and V. S. Pai. SSDAlloc: Hybrid ssd/ram memory management made easy. In *NSDI*, 2011.

[10] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *SIGMOD*, 2011.

[11] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA*, 2011.

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

[13] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan. Grasshopper: an orthogonally persistent operating system. In *Journal of Computer Systems*, volume 7, pages 289–312, 1994.

[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] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.

[16] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, 1981.

[17] P. Helland, H. Sammer, J. Lyon, R. Carr, and P. Garrett. Group commit timers and high-volume transaction systems. In *Tandem TR 88.1*, 1988.

[18] Intel. X-25 mainstream ssd datasheet. http://www.intel.com/design/flash/nand/mainstream/index.htm.

[19] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[20] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.

[21] D. Lowell and P. Chen. Free transactions with rio vista. In *SOSP*, 1997.

[22] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[23] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43:92–105, January 2010.

[24] M. Satyanarayanan, H. Mashburn, P. Kumar, D. Steere, and J. Kistler. Lightweight recoverable virtual memory. In *ACM Transactions on Computer Systems*, 1994.

[25] M. Saxena and M. M. Swift. FlashVM: Virtual Memory Management on Flash. In *Usenix Annual Technical Conference*, 2010.

[26] E. Shekita and M. Zwilling. Cricket: A mapped, persistent object store. In *Workshop on Persistent Object Systems*, 1990.

[27] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: good, fast, cheap persistence for c++. In *SIGPLAN OOPS Mess*, 1993.

[28] M. Vaz Salles, T. Cao, B. Sowell, A. Demers, J. Gehrke, C. Koch, and W. White. An evaluation of checkpoint recovery for massively multiplayer online games. In *VLDB*, 2009.

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

[30] S. J. White and D. J. Dewitt. Quickstore: A high performance mapped object store. In *VLDB Journal*, pages 629–673, 1995.

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