Hathi: Durable Transactions for Memory using Flash

Mohit Saxena, Mehul A. Shah, Stavros Harizopoulos, Michael M. Swift, Arif Merchant

HP Laboratories
HPL-2011-161

Keyword(s):

transactions; transactional memory; flash; durability; consistency; recovery

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 inmemory.

External Posting Date: September 21, 2011 [Fulltext] Approved for External Publication
Internal Posting Date: September 21, 2011 [Fulltext]

Hathi: Durable Transactions for Memory using Flash

Mohit Saxena, *U. Wisconsin-Madison*
Mehul A. Shah, Stavros Harizopoulos, *Hewlett-Packard Labs*
Michael M. Swift, *U. Wisconsin-Madison*
Arif Merchant, *Google*

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, durable updates to in-memory data structures that survive program and system failure.

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. On an initial prototype, we find that Hathi can achieve 1.25 million txns/s with a single SSD.

1 Introduction

Transactional stores are useful in a variety of contexts from internet-facing sites to financial applications. The most ubiquitous transactional stores are relational DBMSs, whose design is based on assumptions from decades ago. They are optimized for spinning disk, single-core CPUs, and traditional enterprise workloads. Today's technology and application landscape, however, is significantly different. As a result, there has been a movement to replace DBMSs with much simpler structured stores, designed from the ground up with modern platforms and applications in mind [6, 10]. We agree that it is time to rethink DBMSs, but argue to keep an essential service they provide: ACID transactions.

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 [5, 11]. Second, transactions provide a simple interface for managing the durability and consistency of application state in the face of failures.

In this paper, we leverage three important recent architectural trends to design a high-performance and ACID-compliant transactional store. 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 [9, 13]. Second, power considerations have driven processor manufacturers away from uni-processors towards multi-core chips. Third, flash-based solid-state drives (SSDs) have become practical and provide 1-2 orders of magnitude lower latency than the fastest disks.

We describe Hathi, 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 [11] with an SSD-optimized write-ahead logging and checkpointing scheme for durability ("D"). In this paper, we focus on two approaches to reduce and eliminate the overhead of persistence.

First, Hathi provides commit options that have not been traditionally used in ACID-compliant data stores to allow developers leverage application knowledge for increased performance. In addition to synchronous and asynchronous commit, which traditional DBMSs also provide, Hathi exposes split-phase and partitioned commit. The split-phase commit decouples the installation of in-memory updates from the flush of 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. Partitioned commit allows applications to cheaply commit transactions that operate on partitioned data with no dependencies across partitions. With these interfaces, applications retain the recoverability guarantees of synchronous commit at speeds closer to asynchronous commit. Initial experiments with these interfaces show 4-5x throughput improvements over synchronous commit.

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.

With these optimizations, initial experiments 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. Moreover, in these experiments, the system scales to many cores and is almost transaction bound. Thus, we provide 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, B-trees, and social-network databases.

2 Background and Related Technologies

In this section, we describe the compelling characteristics of flash drives and discuss how Hathi compares to other main-memory data stores.

NAND Flash Solid-State Drives. Commercial SSDs composed of NAND flash memory are readily available today at reasonable prices. SSDs are internally comprised of multiple flash chips accessed in parallel. As a result, they provide scalable bandwidths limited only by the serial interface between the on-chip register and off-chip controller [4]. In addition, SSDs provide access latencies orders of magnitude faster than traditional disks. For example, the Fusion IO enterprise ioDrives provide 25 μs latencies and up to 600 MB/sec throughput [1]. Consumer grade SSDs provide latencies down to 50-75 μs and bandwidths up to 250 MB/sec. The enterprise SSDs support longer I/O queues and require several outstanding I/O requests to be saturated. With current technology, SSD bandwidth is comparable to or exceeds network bandwidth, and latency is much shorter than user interactions or most network latencies.

Memory-centric Data Stores. Several recent projects propose building transactional interfaces on emerging storage class memory (SCM) such as phase change (PCM), spin-torque-transfer (STT), and memristor [7, 14], which all promise much better performance properties than flash. Hathi seeks the same goal, but achieves it with current technology rather than depending on speculative technologies that may not succeed commercially. Moreover, even though SCMs have much lower latencies, we believe that flash performs *well enough* that they may not be needed: with flash, storage is no longer the bottleneck for many applications.

Closely related to Hathi are other main memory data stores. These fall into three categories: relational stores, e.g. TimesTen and VoltDB [3]; object stores, e.g. GemStone and RamCloud [13]; and key-value stores, e.g. memcached [2]. There are commercial, open-source and research versions of each of them. 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 not on durability methods tuned for Flash. In addition, most of these provide higher-level structured interfaces unlike Hathi, which provides an unstructured memory space. We envision Hathi as a major building block for such systems.

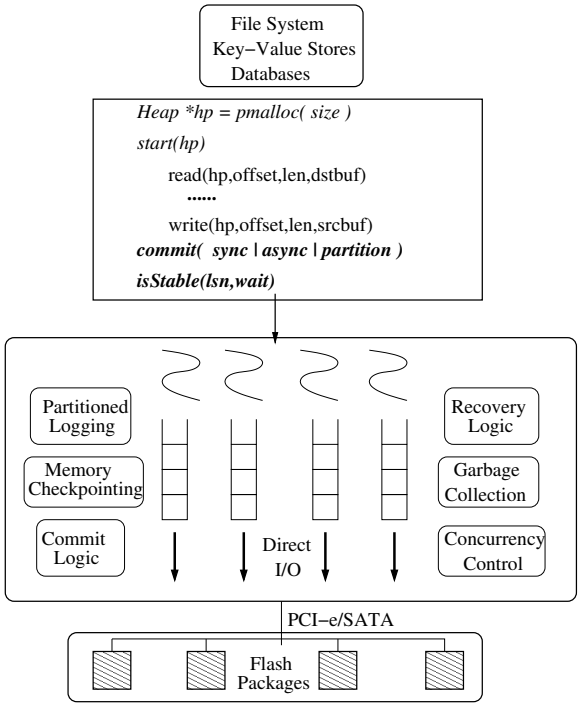


Figure 1: **Hathi Architecture and Interface.**

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.

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:

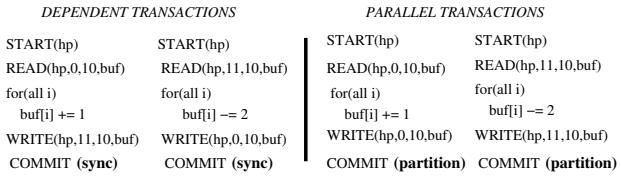


Figure 2: **Dependent and Parallel Transactions.**

sync, *async*, and *partition*.

Sync and Async. *Sync* and *async* are similar to database synchronous and asynchronous commit. Synchronous commit only returns after forcing the transaction's log records and all preceding transactions' records that it depends upon to stable storage, guaranteeing that the update is durable. In contrast, asynchronous commit returns as soon as the transaction finishes updating memory, and does not wait to force the log records to storage. With this option, after a failure, the heap recovers to a consistent point in the transaction history, but 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 knows the thread's transactions are independent of transactions in other threads. In this case, waiting for other threads to flush their preceding transactions is unnecessary. This case arises, for example, when applications partition their data across threads and ensure transactions touch only local data. Such partitioning may be easy when updating regular data structures like a hashtable or a matrix. In this case, isolation is unneeded, so transactions provide atomicity and durability, and the programmer is responsible for consistency. Although applications can mix this option with the others above, they must be careful to ensure the independence of partition commit. Figure 2 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. This interface allows applications to make commit split-phase. They can continue with other tasks and return to dependent actions once the log flushes. In this way, we can overlap I/O and computation, getting recoverability at nearly the same throughput as asynchronous commit.

4 Design and Implementation

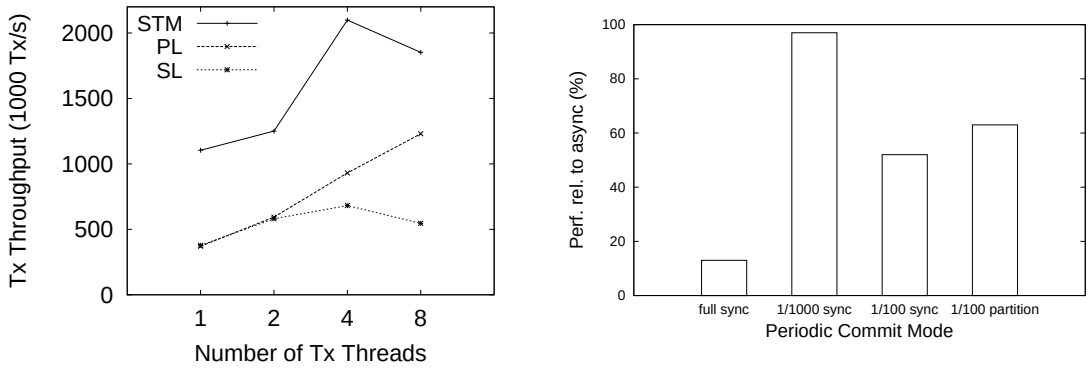
We implemented a prototype of Hathi using TinySTM [8] out-of-the-box for concurrency and added write-ahead logging and recovery for durability. Unlike traditional databases, Hathi does not maintain a backing store: storage contains only logs and checkpoints, and the logs contain only redo records of updates. Checkpoints are copies of the heap state on SSDs that are needed for trimming the logs and reducing recovery time. Hathi incrementally checkpoints the heap to avoid stalling the system.

Hathi's transaction API wraps the underlying STM calls and buffers writes until commit. On successful commit, Hathi tags the writes with a logical sequence number (LSN) given by the STM and inserts them into the 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 well 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. 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.

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*. Partitioned commit annotates the transaction's log record with a *partition* flag, flushes the log,



(a) Durability Costs

(b) Commit Mode Performance

Figure 3: **Durability costs and commit modes' performance.**

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. Finally, `isStable` compares the given LSN against `min_lsn` and waits if necessary.

Checkpointing and Log Trimming. In checkpointing, Hathi periodically writes memory in configurable fixed-sized *chunks* to a non-volatile store. 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. 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 reapplied 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.

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. 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 next to garbage collect.

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*.

5 Preliminary 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 and increased transaction size.

Methodology. We use two setups in these experiments. To emulate a high-throughput OLTP 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 in which each thread continuously executes transactions that read and write six random word offsets in a 4 GB heap. The second system runs the travel reservation workload from STAMP transactional memory benchmark suite [12] 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. Unless otherwise noted, we use asynchronous commit and no checkpointing. We use a 10 ms group commit timer and maximum 512 KB log buffer for each thread.

Durability Costs. Figure 3(a) compares the performance of the OLTP system running with Hathi durability (PL) and without durability (STM). The STM system peaks in throughput at 4 threads with 100% CPU utilization. With Hathi at 8 threads, we reach 1.25 M txns/sec with 70% CPU utilization and saturate only 20% of peak FusionIO bandwidth. This is 38% short of the peak STM only throughput. Even with group commit, our current implementation causes the transaction threads to block

on each log flush. We believe with use of separate logging threads, we could eliminate such blocking and potentially reach the same throughput as pure STM without logging. On the STAMP workload with partitioned logging, we nearly reach 200 K txns/sec, only 15% short of the STM-only throughput. Note that 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 [7, 14].

Partitioned Logging. Figure 3(a) also compares the performance transaction throughput for single log (SL) and partitioned log (PL) on FusionIO. At best, SL achieves only 45% the performance of PL, with only 20% CPU utilization and 10% of peak FusionIO bandwidth, showing that 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. When we repeated these experiments with a traditional SATA disk, the lack of device parallelism and larger transfer costs reduced throughput, for both single and partitioned logs, to one sixth of that using partitioned logging with flash.

We also investigate the impact of logging on the average transaction latency. The average transaction latency for flash is 9 ms, less than the group commit timer of 10 ms for the single log system. For partitioned logging, the extra queuing delay for outstanding I/O requests is compensated by the low access latency of the SSD. With disk, commit latency is 7x higher at over 60 ms, irrespective of the logging strategy.

Performance Tradeoffs. Using the STAMP workload, we investigate the impact of different durability options by mixing synchronous and partitioned commit with asynchronous commit in different proportions. Figure 3(b) compares the performance of fully asynchronous transactions with fully synchronous. As expected, throughput drops by 87% because of the frequent stalls for I/O to complete. However, when an application can commit most transactions asynchronously, illustrated by making 1 in 1000 synchronous, it drops by only 3%. We also compare the performance of synchronous and partitioned commit by making 1 in 100 transactions synchronous or partitioned. For 1/100 sync and 1/100 partition, performance is 50% and 40%, respectively, of fully asynchronous. Thus, 1/100 partition is about 25% faster than 1/100 sync. Clearly, periodic synchronous and partitioned commit provide a middleground between the performance of asynchronous commit and the recoverability of synchronous commit.

Finally, we find that increasing transaction sizes increases both code path length and STM contention. For example, 8 concurrent threads reading/writing 512 bytes provide half the transaction throughput of 4 threads.

6 Discussion

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 argued the case for 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. Finally, we envision Hathi as a platform that can be used to build not only user-facing applications, but also scalable and robust infrastructure.

References

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

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

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

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

[5] 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.

[6] 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.

[7] 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.

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

[9] R. Kallman et al. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[10] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

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

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

[13] 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.

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