Reducing Synchronization Overhead Using Hardware Transactional Memory

Vamsi Chitters
vchitters@berkeley.edu

Adam Midvidy
amidvidy@berkeley.edu

Jeff Tsui
tsui.jeff@berkeley.edu

ABSTRACT

Transactional Memory (TM) is a high-level abstraction for synchronizing access to shared data that allows programmers to easily group shared data accesses into atomic blocks. Intel's latest processor generation, Haswell, provides support for the Transactional Synchronization Extensions (TSX), an extension to the x86 instruction set that provides TM functionality. We modify LevelDB, a write optimized persistent key-value store, to use TSX for synchronization. A comparison of the TSX-augmented LevelDB with the original LevelDB is performed to assess the performance tradeoffs of TSX under various workloads. Our modified LevelDB exhibits increased performance on write-intensive simulated workloads, as well as 25% improvement in write throughput on the standard LevelDB benchmark suite.

Keywords

LevelDB, Transactional Memory, Intel, Haswell, TSX

1. INTRODUCTION

As physical limitations have curtailed the growth of processor clock speeds, CPU designers have turned to increasing the number of processor cores on a single CPU to improve performance. Consequently, application developers must exploit concurrency in their programs to fully utilize the underlying hardware. Unfortunately, traditional lock-based approaches to coordinating access to shared data lead to an unappealing tradeoff between performance and complexity: coarse-grained locking results in increased synchronization overhead, while fine-grained locking imposes a significant mental burden on the programmer, resulting in race conditions, deadlock and starvation.

Taking a page from the database research community, systems researchers have proposed Transactional Memory (TM) as a higher-level abstraction for synchronizing access to shared data. With TM, programmers group related updates to shared data into atomic blocks that are executed as an atomic unit, in isolation from other processes or threads that may concurrently access the shared state. Programmers no longer have to keep track of locks and the complexities associated with them (deadlock, starvation, and priority inversion etc.) nor do they have to worry about the overhead of lock handling.

```
def transferCoarse(from_acct, to_acct,
    amt):
    globalAccountLock.lock();
    from_acct -= amount;
    to_acct += amount;
    globalAccountLock.unlock();
```

Listing 1: Coarse grained synchronization

```
def transferFine(from_acct, to_acct,
   amt):
   if from_acct.id < to_account.id{
       from_acct.lock()
       to_acct.lock()
   } else {
       to_acct.lock()
       from_acct.lock()
   }
   from_acct -= amount;
   to_acct += amount;
   from_acct.unlock()
   to_acct.unlock()
```

Listing 2: Fine grained synchronization

```
def transferTM(from_acct, to_acct,
  amt):
  atomic {
      from_acct -= amount;
      to_acct += amount;
  }
```

Listing 3: Synchronization with TM

This paper is a research project on Hardware Transaction Memory for CS262A. December 2013, Berkeley, California.

To further compare transactional memory to traditional database transactions, atomic blocks allow programmers to enforce atomicity, consistency, and isolation in the synchronized region of code. A transactional block either commits successfully or aborts and restores pre-transactional state, the very definition of atomicity. If the transactional blocks are grouped correctly with regards to the semantics of the application, all executions will result in a consistent state. Furthermore, concurrent transactions are isolated from each other as uncommitted results are stored in a thread local L1$.

Transactional memory can be implemented at the hardware (HTM) or software (STM) level. STM optimistically updates shared data while keeping track of a log of all modifications of shared state. Transactions are later validated by inspecting the log and verifying that there are no concurrent accesses to shared data before committing. STM suffers from significant performance in maintaining the log and verifying transactions when compared to fine grained locking. On the other hand, HTM leverages the underlying CPU's cache and bus protocol to support transactions. HTM has previously been implemented in experimental architectures, such as the Sun Rock processor [8], specialized mainframe processors such as the IBM Blue Gene/Q [13] and a specialized Java acceleration appliance by Azul Systems [4]. However, Intel's recent Haswell architecture includes the first HTM implementation widely available on a commodity architecture. As HTM heads towards being widely available on commodity servers, multithreaded applications must exploit the advantages of HTM to make full use of the underlying hardware.

2. BACKGROUND

2.1 Lock Elision

Intel's latest processor architecture, Haswell, supports an extension to the x86 instruction set called the Transactional Synchronization Extensions (TSX). TSX implements *best effort* hardware transactions, that is, transactions are not guaranteed to commit and a non-transactional fallback path must be specified to ensure forward progress. In practice, real-world code employing TSX must maintain a separate lock-based code path, with hardware transactions providing a potential performance increase due to the ability to perform *lock elision*. Lock elision is an approach to speed up existing lock-based programs with TM by allowing threads to speculate past critical section locks in an optimistic manner, using the underlying transactional runtime to detect data conflicts and retry aborted transactions. As explained by by Roy, et. al [3]:

In general, lock elision can only improve scalability when (a) there is little contention between the work different threads do in a critical section (e.g. speculation cannot speed up a critical section incrementing a single shared counter), and (b) there is contention for the lock protecting the critical section.

2.2 TSX: Mechanical Sympathy

In the current implementation of TSX, when a hardware thread begins transactional execution, all writes to memory by that thread are buffered in L1$. Note that this limits the size of data that can be modified in a single transaction to the L1$ size (32KB, 8-way set associative), which is further reduced if hyperthreading is enabled, as two threads will then share a single L1$. Aborts that occur due to insufficient buffer space are termed *capacity aborts*. When a transaction commits, the processor uses the cache-coherency protocol to check for conflicting data access by other threads at the granularity of a single cache line. If there are no conflicts, a thread commits its writes to main memory, at which point modifications made during the transaction become visible to other threads. Aborts that occur due to conflicting data modifications are termed *conflict aborts*. If the transaction aborts due to conflict or capacity reasons, register and memory state are restored to their contents at the beginning of the transaction. In addition to capacity and conflict aborts, TSX transactions can be aborted for additional reasons including any event that causes a switch to the kernel (page faults, interrupts, system calls, etc.) and certain 'unfriendly instructions' such as PAUSE or instructions to set the direction of the branch predictor.

Intel provides two interfaces to TSX: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). HLE instructions are backwards compatible with older architectures and can be used elide existing spinlocks in existing applications without modifying code, but HLE mandates an inflexible abort path that defaults to acquiring the lock after a single transactional abort. The RTM instructions are not backwards compatible, but they allow the programmer to specify an alternate fallback path to be executed if a transaction aborts. In this paper we elect to utilize the more powerful RTM instructions as backwards compatibility is not a concern and RTM's increased flexibility permits greater variety of optimization techniques.

2.3 LevelDB

We aim to use the new HTM capabilities in Haswell to improve the performance of LevelDB [18], a write optimized, persistent key-value store. Google developed LevelDB as an embedded key value store based on the BigTable[9] SSTable structure, which provides a persistent, ordered immutable map from byte-string keys to values. LevelDB was released as a 'clean' implementation of the SSTable design, without dependencies on other proprietary Google technologies. As a result of its high quality codebase and permissive open-source BSD license, LevelDB has been adopted as a building block for many other open source projects. Prominent ex-

amples include distributed databases such as HyperDex[12] and Riak[23], the IndexedDB API in Google Chrome and the official Bitcoin client[6].

The on disk storage format of LevelDB is based on the log-structured merge-tree (LSM tree) [20]. This provides indexed access to files with high insert volume and can be used to increase write throughput. The core idea behind this form of memory management is to exploit the tradeoff between size per level and data access rate. As data fills up the closest level to memory, a compaction process takes place and moves the data to the next level where there is more space, but increased cost to access the data. As a result, read access is slow when there are multiple levels to search for to find the specified key.

Stock LevelDB uses coarse grained locking for all write transactions. A write operation in LevelDB consists of acquiring a mutex, writing the data to memory buffer in a skip list data structure (similar to a B-tree) and recoverable log, then releasing the mutex. During the period where a thread holds the mutex, no other write threads can access the data. Once the data is in memory, a background process arranges the data in a hierarchy of levels based on access frequency. When the skip list is filled, the data is moved to immutable memory, and when immutable memory is filled it is compacted to an ordered array of levels in the form of .sst files.

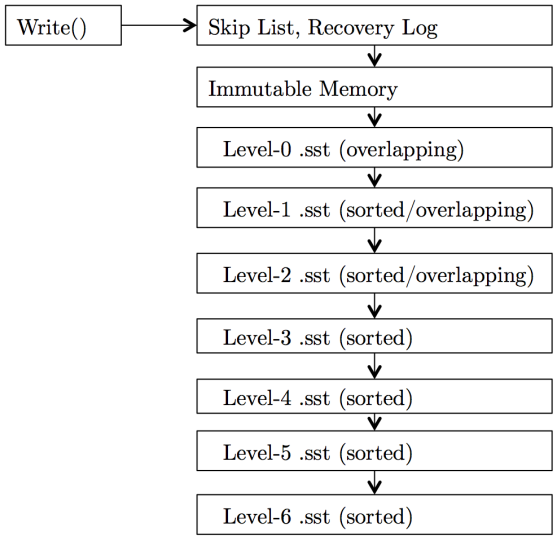

Figure 1: LevelDB Write Path - Modified from Basho Technology speaker deck [5]

We choose to modify LevelDB (as opposed to other databases or applications) because of its high quality codebase and its usage of coarse grained locking. LevelDB keeps a global mutex that is acquired during all operations to ensure that modifications to in-memory data structures are atomic. This concurrency scheme presents a good opportunity for lock elision as one expects that eliding a shared mutex will lead to higher throughput if concurrent threads operate on disjoint data.

3. INITIAL APPROACH: SYNTHETIC BENCHMARK

To get an understanding of the performance characteristics of TSX we wrote a preliminary benchmark to measure the overhead of transactional execution relative to that of fine and coarse grained locking. Each synchronization method was used to guard an array of counters that were randomly incremented 1 million times each by 8 threads. At the conclusion of the benchmark, the counters were summed to verify that the correct number of writes was recorded. Figure 2 shows the resulting write throughput for each synchronization method.

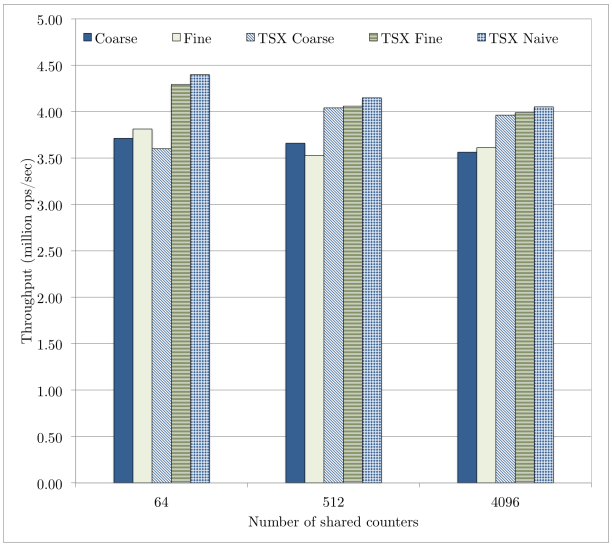

Figure 2: Synchronization overhead as a function of contention. As the number of elements increases, the probability of two threads accessing the same resource decreases. TSX provides higher throughput over a wide range of contentions compared to coarse and fine grain locking.

For the coarse grained implementation, all threads competed to acquire a shared spinlock before modifying a counter. The fine grained implementation used a separate spinlock for each entry. For the naive TSX implementation the write was wrapped in a transaction that would retry indefinitely until committing. While this method gave the best performance as the transactions were short and usually committed successfully, it is not possible to use TSX in this manner in a production system as forward progress is not guaranteed. The TSX enabled implementations reused the same coarse and fine grained locking code, but elided the actual lock acquisition using an XScope wrapper (see section 4.1). We also enabled transaction coarsening (section 4.2.2) and adaptive lock elision (section 4.2.1). Note that the TSX accelerated versions have higher throughput for most contention

levels. For the high contention case with 64 counters, the TSX-coarse version performs badly as the high rate of aborts causes the fallback path to be executed frequently. As the amount of counters increases, thereby decreasing contention, the performance of the TSX-coarse version approaches the performance of the TSX-fine grained implementation (which is faster for all contention levels).

4. IMPLEMENTATION

4.1 XSync

LevelDB provides cross-platform mutex and condition variable classes that delegate to the underlying system's native synchronization facilities. On the Linux system used for development, the underlying primitives were provided by the PThreads API. To avoid cluttering the LevelDB codebase with raw TSX instructions, we decided to implement TSX-enabled versions of LevelDB's synchronization primitives.

Rather than explicitly locking a shared mutex upon entering a critical section, LevelDB employs the scoped locking pattern, in which a stack allocated object is created to acquire the lock for the duration of a given scope. We developed `XScope`, a TSX-enabled lock wrapper that makes it simple to elide locks in code that already uses scoped locking. When an `XScope` object is instantiated, the thread attempts to execute transactionally for the lifetime of the containing scope. If an abort occurs, the transaction is retried until a predetermined limit of aborts is reached, at which point the lock is acquired and the scope is re-executed non-transactionally.

```
// Unsynchronized
doThreadSafe();
{
    // mu_ is a shared mutex
    MutexLock lock(&mu_);
    // Synchronized by lock
    doCritical();
}
// MutexLock destructor is called
// when the previous scope
// exits, so execution is currently
// unsynchronized
```

Listing 4: Example LevelDB MutexLock construct

```
// Unsynchronized
doThreadSafe();
{
    // mu_ is a shared mutex
    xsync::XScope xact(&mu_);
    // Synchronized by TSX
    // (or possibly lock if
    // there have been many aborts)
    doCritical();
}
```

Listing 5: Lock elision with XSync

In addition to mutexes, LevelDB also makes use of condition variables to order concurrent writes in a fair manner. The stock codebase provides a condition variable class that wraps Pthread condition variables; unfortunately, any operations on the stock condition variables cause aborts during transactional execution as they result in the execution of system calls (see section 2.1 for an overview of TSX abort causes). As multiple condition variable operations are used in the LevelDB write-path, we implemented `XCondVar`, a deferred-signal transactional condition variable [21] that does not cause aborts during transactional execution.

Our implementation relies on the Linux futex abstraction, which provides a low level interface to kernel synchronization facilities. The `futex` wait call takes as input a pointer to an integer in memory (the '`futex` counter') that is shared between the threads using that condition variable and a comparison value. The call returns immediately if the value of the `futex` counter is different than the comparison value, otherwise the thread is put to sleep and added to a queue in the kernel. There is also a corresponding signal call that takes the futex counter as input as well as the number of threads to wake from the queue. For additional performance, we use an undocumented kernel flag that provides faster operation in the case that the futex is shared only between threads of a single process.

When a thread executes a *wait* operation on an `XCondVar`, it first commits partial results, and then waits on a `futex` until it is awoken by a signaling thread, at which point it resumes transactional execution. We exploit the `futex` counter mechanism to prevent lost wakeups due to concurrent signal and wait calls. The current value of the counter is read by the waiter before committing partial results, and then the cached counter value is used as the comparison value for the subsequent wait call. A signaling thread performs an atomic increment of the `futex` counter before executing the signal call. Thus a waiting thread will immediately return if another thread signals between the time at which a waiter begins committing partial results and when the `futex` wait call is actually executed.

The `XCondVar` signal operation is implemented by registering a callback that is executed by the signaling thread after it commits its current transaction. As TSX does not natively provide commit actions, we added runtime support for them in the `XScope` wrapper. An `XScope` instance maintains an internal queue of callbacks that is executed in FIFO order when the transactional scope exits. We note that the recent addition of powerful C++11 features such as `std::function`, `std::bind`, and inline lambda expressions enabled a succinct implementation of this feature.

4.2 Optimizations

We explored various optimization techniques, particularly within the local synthetic benchmark, which are worth mentioning with respect to hardware transactional memory.

4.2.1 Adaptive Lock Elision

Robust usage of TSX requires a separate a fallback path to be available in the case that the transaction repeatedly aborts. Furthermore, there must be some kind of retry policy in place to determine when the fallback path should actually be taken. The simplest policy is to use a static limit to the number of attempts to execute the critical section transactionally. While it is possible to optimize this limit carefully when the workload is known in advance, the static approach is suboptimal when the amount of contention is unpredictable. To better exploit the performance of TSX in such situations we developed adaptive lock elision policies that scale the number of retries dynamically at runtime.

To perform adaptive lock elision, each thread maintains a vector of statistics counters to record the number of total aborts, the number of successive aborts, the number of successive commits and the number of total commits. The counters must be thread-local as they are incremented during transactional execution, and they are padded to a cache line to reduce the overhead in L1$ usage and to minimize cache misses. When a thread aborts a transaction, we immediately reduce the retry limit to one, to rapidly reduce the overhead of failed lock elision during periods of high contention. When a transaction commits successfully, the abort count limit is incremented up to a predefined maximum. This technique allows lock elision to perform effectively under varying levels of contention.

4.2.2 Transaction Coarsening

During periods of low contention, it can be profitable to combine multiple transactions into a single larger transaction to amortize the overhead of entering and exiting a transactional region. Intel researchers [24] have termed this technique *transaction coarsening*, which they perform by statically merging the transactions of separate loop iterations during a tight loop over a critical section. In line with our approach to lock elision, we prefer a coarsening policy that can adapt to the level of contention at runtime. As relevant statistics are already kept to perform adaptive lock elision, we piggyback off the same thread-local counters to perform transaction coarsening when low contention is detected at runtime.

We initiate transaction coarsening when a thread successfully reaches a predetermined threshold of successive commits. When the active XScope exits, the aforementioned coarsening criterion is tested, in which case the scope exits without committing the current transaction. When the thread subsequently starts a new transaction, we branch on the xtest instruction [14] , returning immediately if the thread is already executing transactionally. If a thread successfully commits a coarsened transaction, it will attempt to execute its next transaction at an even larger granularity. However, as transaction coarsening greatly increases the required L1$ buffer space needed to store uncommitted results (section 2.2), we halve the transaction granularity in response to any capacity aborts. If any conflict aborts occur, we disable coarsening and reexecute the transaction at its natural granularity.

5. EVALUATION

In this section, we evaluate our accelerated LevelDB with Intel TSX against the stock LevelDB on a set of synthetic and simulated application-level workloads. Micro-benchmarks are performed using LevelDB's built in db_bench benchmarking suite and macro-benchmarks are done through Yahoo! Cloud Serving Benchmark (YCSB) [7].

Our evaluation answers the following questions:

1. How does the performance of LevelDB modified with TSX compare to stock LevelDB?
2. What kinds of workloads benefit most from using TSX?
3. What are the performance tradeoffs for hardware transactional memory?

5.1 Methodology

All our evaluations are performed on a single machine with the an Intel Haswell i7-4770 CPU supporting TSX intrinsics. The machine has 4 physical cores running at 3.40 GHz and supports 8 logical threads with hyperthreading enabled. The total cache size is 8MB, with an L1$ cache size of 32KB. Our evaluation focuses on write-heavy workloads as LevelDB's log structured merge tree architecture is explicitly designed for high write performance. The micro-benchmark focuses exclusively on writes while the macro-benchmark using YCSB comprises both a write heavy workload as well as six other workloads that cover a wide range of simulated real world use cases to assess the overall performance tradeoffs of TSX modified LevelDB.

5.2 LevelDB Benchmark Suite

We first measure the performance of the included benchmarks from LevelDB's db_bench suite. The first measures sequential writes (FillSequential) and the second measures random writes (FillRandom). In each benchmark, 100,000 key-value pairs are inserted into LevelDB using varying numbers of threads and the resulting throughput is measured. With a higher number of threads, the amount of contention increases in the workload due to the need to modify shared data-structures during the LevelDB write path. Both benchmarks use 16 byte keys, and 50 byte values. These micro-benchmarks show a 20-25% increase in throughput with TSX under mid-high contention workloads.

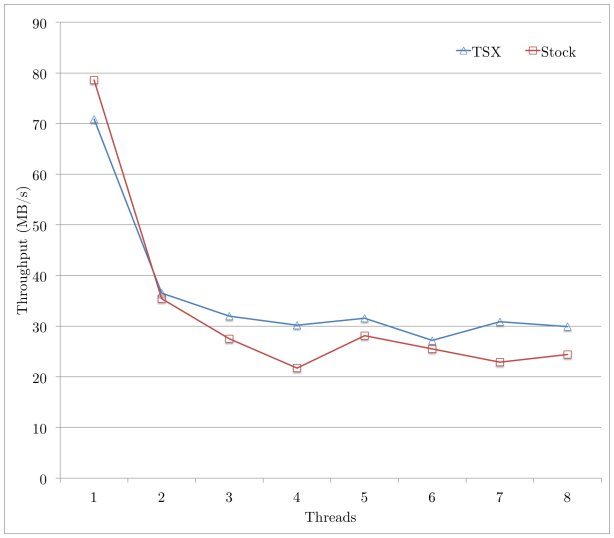

Figure 3: Throughput for 100,000 sequential inserts done with varying numbers of threads using LevelDB's FillSeq benchmark.

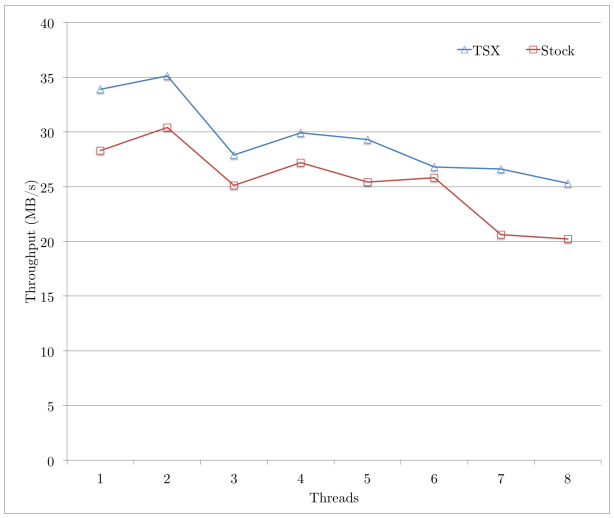

Figure 4: Throughput for 100,000 random inserts done with varying numbers of threads using LevelDB's FillRandom benchmark.

The FillRandom benchmark shows throughput improvement with respect to stock LevelDB as the number of threads varies from 1 to 8. At 8 threads, we see a 25.2% improvement in overall throughput. Similarly, the FillSequential shows a higher performance for the TSX accelerated LevelDB, with a 22.5% performance increase with 8 writer threads. We posit that the contention in the system is higher for the FillRandom benchmark due to the need to sort a value in the in-memory before completing a write. As LevelDB releases the database lock when data is actually being read or written to disk, the writers in the FillRandom benchmark collectively spend more time holding the lock and thus the FillRandom benchmark benefits more from the reduced synchronization overhead provided by lock elision.

The two LevelDB write benchmarks provide a good understanding of how TSX interacts with LevelDB. Clearly TSX results in better throughput as a result of eliding locks and eliminating locking overhead. As the amount of contention increases, there is a greater performance improvement with TSX modified LevelDB compared with stock LevelDB.

5.2 Yahoo! Cloud Serving Benchmark (YCSB)

In addition to the performance tests described above, we evaluate TSX modified LevelDB using a macrobenchmark to gauge its performance characteristics on application-level workloads. This allows us to understand TSX implications on LevelDB when stress-tested with real-world data. We use the Yahoo! Cloud Serving Benchmark (YCSB), an industry-standard benchmark for cloud storage performance. YCSB is easy to extend and configure, providing many tunable parameters to express a variety of workloads. Each workload is configured by specifying the number of operations to perform, the ratio of reads, inserts, updates, and scans, number of threads simultaneously executing the operations, and the type of distribution used to access the data.

To test out the hypothesis that there will be noticeable performance benefit with TSX on write heavy workloads, we create a new write heavy workload of 10 million operations with 50% reads and 50% inserts (values are 1 KB in size), using a zipfian distribution for data access. A zipfian distribution simulates real world data access patterns by taking into account the fact that some records are more popular (head of the distribution) while the majority are not (the tail). The access pattern of 50% reads and 50% writes is chosen to mimic the workload experienced by a typical session store keeping track of user state such as cookies. We measure the throughput and latency for TSX modified LevelDB and compare it to that of stock LevelDB for varying numbers of threads.

TSX modified LevelDB exhibits considerably better throughput than stock LevelDB while running the workload using 8 threads. Figure 5 shows the difference in the number of operations completed by TSX minus the number of operations completed by stock LevelDB each minute. The fluctuations in the graph are due to different write compaction intervals, but the overall trend is clearly increasing for 2 and 8 threads. These cases confirm our expectations and are consistent with the db_bench results that LevelDB provides performance improvements for write heavy workloads with contention. We are still investigating why stock LevelDB completes more operations than TSX modified LevelDB for the majority of the run time for the 4 thread case.

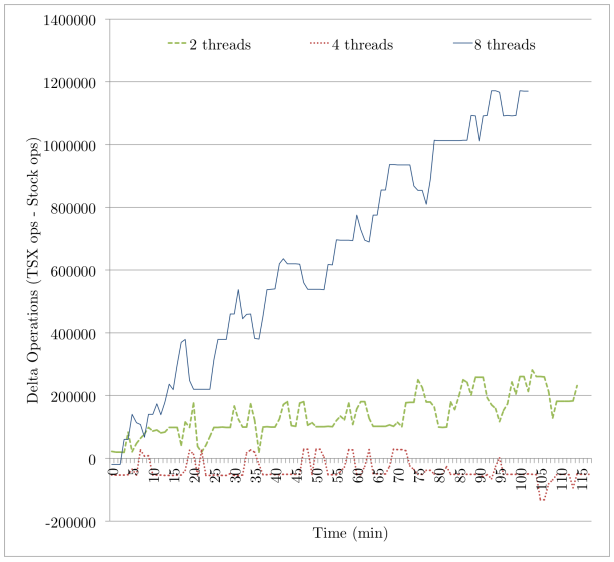

Figure 5: Difference in operations completed over time (TSX operations - Stock Level DB operations) sampled at each minute. The workload consists of 10 million operations with 50% reads and 50% writes using a zipfian distribution.

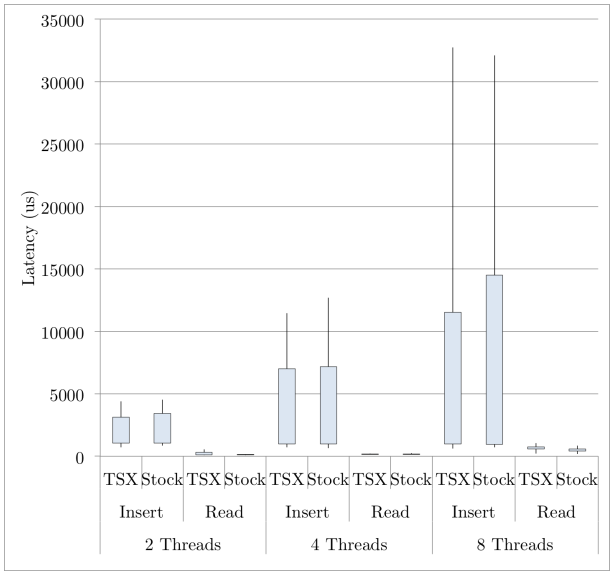

Figure 6: 10, 25, 75, and 90th percentile latencies vs number of threads for a workload of 10 million operations with 50% reads and 50% writes using a zipfian distribution.

We also measure the latency associated with reads and inserts for this workload for 2, 4, and 8 threads (Figure 6). While the read latencies are relatively equal for TSX modified and stock LevelDB, there is significant write latency improvement for TSX modified LevelDB. Furthermore, the amount of improvement increases as the contention in the workload increases. In the 8 thread case, TSX modified LevelDB decreases the latency the most, as seen by the lower 75th percentile. This is consistent with our understanding of the coarse grained locking characteristics of LevelDB and shows there is a significant amount of time spent in waiting for locks (locking overhead). The high amount of variation in latencies is primarily due to write compactions. When one of the .sst files in LevelDB becomes too full, the data is compacted and moved to a lower level. During these write compactions, none of the operations (reads or inserts) are able to proceed. While these write compactions can not be avoided using lock elision, TSX effectively reduces the latency for inserts, which is the primary source of latency in LevelDB.

Workload	Operations	Record Selection	Application Example
A – Update heavy	Read: 50% Update: 50%	Zipfian	Session store
B – Read heavy	Read: 95% Update: 5%	Zipfian	Photo tagging
C – Read only	Read: 100%	Zipfian	User profile cache
D – Read latest	Read: 95% Insert: 5%	Latest	User status Updates
E – Short ranges	Scan: 95% Insert: 5%	Latest/Uniform	Discussion group threads

Figure 7: The five core workloads that are part of YCSB's benchmark suite–standard metric used by various other projects.

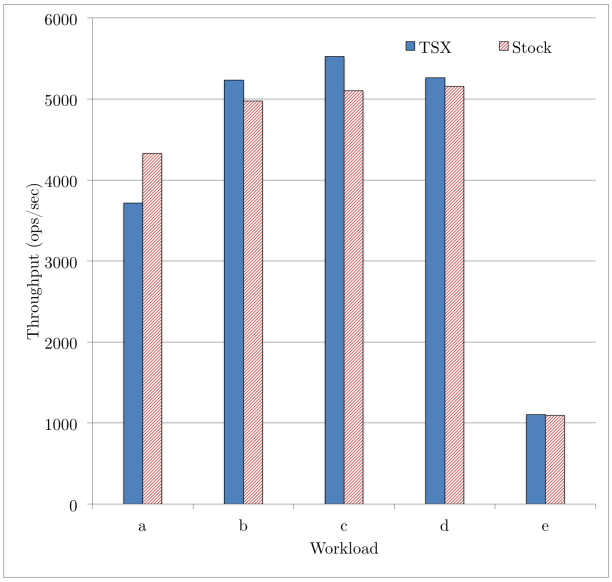

Figure 8: Throughput for the 5 core workloads specified in the Yahoo! Cloud Serving Benchmark designed to simulate the spectrum of real world workloads. The workloads were modified to run with 4 concurrent client threads.

YCSB provides a set of 5 core workloads designed to simulate the spectrum workloads experienced by real world systems (Figure 7/8). The workloads are for 1000 operations with varying ratios of insert, read, update, and scan operations. We modify these workloads to perform the same operations across 4 threads to assess TSX performance benefits. TSX has comparable throughput for workload e, lower throughput for work-

load a, and higher throughput for workloads b, c, and d. Overall, however, the throughputs are comparable. We are still investigating why there is not a more measurable performance difference between TSX modified and stock LevelDB. We expected TSX to exhibit higher throughput for workload a and slightly higher throughputs for the others. An explanation may be that the workloads involve much fewer operations compared to the other workloads analyzed and so the results exhibit higher variance. In addition, because we modify the workloads to use 4 threads, this may be lowering the accuracy of the benchmark due to the way YCSB implements multithreaded workloads.

6. RELATED WORK

6.1 Intel's TSX Evaluation

Intel researchers performed an evaluation of TSX performance [24] based primarily on HPC workloads. They were able to achieve an average speedup of 1.41x for HPC workloads, and 1.31x improvement on a user level TCP-IP stack. Our approach differs in that we chose to optimize OLTP-style workloads, a common case in non-HPC distributed computing. While we drew inspiration from the optimizations proposed by the Intel researchers, such as transaction coarsening, we extended their techniques by implementing a lightweight runtime to maintain thread-local statistics to allow runtime adaptation.

6.2 Transactional LuaProc

Skyrme and Rodriguez [1] ported the luaproc coroutine library to use STM for synchronization. Unlike our LevelDB fork, which uses TSX-enabled implementations of traditional synchronization constructs, their luaproc implementation echewed locks and condition variables for unbounded transactions and non-blocking coroutine communication channels. Unfortunately, their implementation resulted in a considerable performance penalty when compared to the existing lock-based version of luaproc - the transactional implementation performed up to 20% worse on average with 8 active threads.

6.3 Transactional Main-memory Databases

Leis et. al propose a new design for a main-memory database optimized to exploit HTM for performance. [28] Their design uses a timestamp ordering scheme to assemble smaller, physical hardware transactions into larger, logical database transactions. The authors achieve a 9x performance increase on a TPC-C benchmark on a *simulated* 32-core HTM capable processor. While the authors claim a greater performance benefit, they also benefit from designing their database system from the group up to exploit HTM; conversely LevelDB was designed to use traditional lock-based synchronization. Furthermore, the authors express their final results in terms of simulated performance on a hypothetical processor, while our evaluation is based on execution speed on a physical CPU.

6.4 Transactional Programming Languages

Brian Carlstrom et. al from Stanford's Computer Systems Laboratory propose a transactional programming language, called Atomos, which is derived from JAVA and allows for a high level approach to effectively use transactional memory. [4] It features atomic blocks that define basic nested transactions, transaction based condition waiting (similar to conditional critical regions), open nested transactions, and violation handlers which handle general exceptions. The performance of Atomos transactional programming is compared to Java lock-based programming and Atomos shows linear scaling up to 32 CPUs and either retains the basic scalability of Java implementation or outperforms it in general (2-5x). Clearly, there are basic principles and implementation ideas that are noteworthy and considered for extension, but the scope of the Atomos paper pertains to software transactional memory techniques, while the goal of this paper is to explore the relatively new area of hardware transactional memory.

7. CONCLUSION AND FUTURE WORK

This paper analyzes the impact of Haswell's TSX instructions on LevelDB. We compare the impact of using lock elision in TSX modified LevelDB versus coarse grained locking in stock LevelDB to show that transactional memory results in performance gains under a subset of workloads. We measure a 20-25% increase in throughput for TSX modified LevelDB for write-only workloads as well as an increase in throughput and latency for workloads consisting of 50% reads and 50% writes. Another major contribution in this project is XSync, a library containing TSX-enabled versions of LevelDB's synchronization primitives. XSync is written to be integrated into LevelDB with minimal effort and we expect the design to be useful in integrating TSX instructions to other systems as well. To fully leverage the performance benefits of TSX, the workload characteristics must be well understood. For the workloads specified in this paper, TSX gives programmers the benefit of easily marking code regions to be executed synchronously, while also providing performance improvements.

As LevelDB is commonly used as a database for a higher-level distributed store, we would like to assess the performance of a TSX optimized LevelDB as a storage engine for a distributed system. Furthermore, there have been many other forks of LevelDB that have claimed to acheive improved performance such as HyperLevelDB [12] and Facebook's RocksDB [26]. We would like to explore applying our modification to those LevelDB forks to see if there could be a further performance gain over the original codebase. Additionally, it remains to be seen if other categories of complex software, e.g. web browsers, compilers, etc, could benefit from transactional memory and/or lock elision.

8. ACKNOWLEDGMENTS

We would like to thank John Kubiatowicz and Anthony Joseph for guiding us throughout the project. In addition, we would like to thank Matt Moskewicz for providing us with access to the latest Haswell machine.

9. REFERENCES

1. Alexandre Skyrme and Noemi Rodriguez. From Locks to Transactional Memory: Lessons Learned from Porting a Real-world Application. In Proc. of *TRANSACT*, 2013.
2. Amitabha Roy, Steven Hand, and Tim Harris. A Runtime System for Software Lock Elision. In Proc. of *EuroSys*, 2009.
3. Arrvindh Shriraman and Sandhya Dwarkadas. Refereeing conflicts in hardware transactional memory. In Proc. of *ICS*, 2009.
4. Austen McDonald, Hassan ChaïňĄ, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The Atomos transactional programming language. In Proc. of ACM SIGPLAN Notices. Vol. 41. No. 6. ACM, 2006.
4. Azul Systems. http://www.azulsystems.com/.
5. Basho Technologies. Optimizing leveldb for Performance and Scale. https://speakerdeck.com/basho/optimizing-leveldb-for-performance-and-scale-ricon-east-2013.
6. Bitcoin. https://github.com/bitcoin/leveldb/.
7. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In Proc. of *SoCC*, 2010.
8. Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum, and Marek Olszewski. Early Experience with a Commercial Hardware Transactional Memory Implementation. Intel, Technical Report TR-2009-180, 2009.
9. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In Proc. of *OSDI*, 2012.
10. Hillel Avni, Nir Shavit, and Adi Suissa. Leaplist: Lessons Learned in Designing TM-Supported Range Queries. In Proc. of *PODC*, 2013.
11. Hubertus Franke and Rusty Russell. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In Proc. of *Ottawa Linux Symposium*, 2002.
12. HyperDex. http://hyperdex.org/.
13. IBM Blue Gene/Q. http://www-03.ibm.com/systems/technicalcomputing/solutions/bluegene/.
14. Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual.* http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html
15. Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual.* http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
16. Jared Casper, Tayo Oguntebi, Sungpack Hong, Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. Hardware Acceleration of Transactional Memory on Commodity Systems. In Proc. of *ASPLOS*, 2011.
17. Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance Pathologies in Hardware Transactional Memory. In Proc. of *ISCA*, 2007.
18. LevelDB. https://code.google.com/p/leveldb/.
19. Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In Proc. of *ASPLOS*, 2011.
20. Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). In *Acta Informatica.*
21. Polina Dudnik and Michael M. Swift. Condition Variables and Transactional Memory: Problem or Opportunity? In Proc. of *TRANSACT*, 2013.
22. Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In Proc. of *International Symposium on Microarchitecture (MICRO)*, 2001.
23. Riak. http://basho.com/riak/.
24. Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance Evaluation of Intel R Transactional Synchronization Extensions for High-Performance Computing. In Proc. of *IEEE - 2013 SC*, 2013.
25. Robert Escriva, Bernard Wong, and Emin Gun Sirer. HyperDex: A Distributed, Searchable Key-Value Store. In Proc. of *SIGCOMM*, 2012.
26. RocksDB. http://rocksdb.org/.
28. Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. In Proc. of *ICDE*, 2014.