Applying HTM to an OLTP System: No Free Lunch

David Cervini
École Polytechnique
Fédérale de Lausanne
david.cervini@epfl.ch

Danica Porobic
École Polytechnique
Fédérale de Lausanne
danica.porobic@epfl.ch

Pınar Tözün*
IBM Almaden
Research Center
ptozun@us.ibm.com

Anastasia Ailamaki
École Polytechnique
Fédérale de Lausanne
anastasia.ailamaki@epfl.ch

ABSTRACT

Transactional memory is a promising way for implementing efficient synchronization mechanisms for multicore processors. Intel's introduction of hardware transactional memory (HTM) into their Haswell line of processors marks an important step toward mainstream availability of transactional memory. Transaction processing systems require execution of dozens of critical sections to insure isolation among threads, which makes them one of the target applications for exploiting HTM.

In this study, we quantify the opportunities and limitations of directly applying HTM to an existing OLTP system that uses fine-grained synchronization. Our target is Shore-MT, a modern multithreaded transactional storage manager that uses a variety of fine-grained synchronization mechanisms to provide scalability on multicore processors. We find that HTM can improve performance of the TATP workload by 13-17% when applied judiciously. However, attempting to replace all synchronization reduces performance compared to the baseline case due to high percentage of aborts caused by the limitations of the current HTM implementation.

1. INTRODUCTION

For decades, processor vendors have increased performance of uniprocessors by increasing their frequency and complexity of a core with instruction-level parallelism, out-of-order execution, etc. Around 2005, however, they hit the frequency scaling wall due to power and thermal limitations, and started increasing performance by placing multiple simpler cores on the same chip. These cores share processor caches and memory controllers. In the past few years, we have seen a steady increase in the number of cores on a chip, as well as in the number of chips in a server.

The abundant parallelism that is present in modern servers poses significant challenges for software systems that require efficient synchronization in multithreaded programs. Synchronization approaches can be divided into two broad categories: lock-based and lock-free. Locks are appealing because they enable the use of the same locking primitive in a number of situations; many locking primitives have interchangable interfaces. However, they require the execution of critical sections. On the other hand, lock-free approaches rely on hardware primitives, e.g., atomic compare-and-swap, and thus can potentially achieve very good performance and scalability. The downside of lock-free synchronization is the requirement for a new synchronization algorithm or data structure for each use case. Transactional memory [4] promises to make lock-free programming easier using atomic read-modify-write operations over multiple memory locations. Software implementations have been an active area of research for years [3], however, they didn't reach mainstream popularity.

Online Transaction Processing (OLTP) is one of the most important and demanding database applications. Traditionally, OLTP workloads run on high end servers with large number of processor cores. Exploiting such abundant parallelism is of utmost importance. However, scaling up transaction processing systems on multicores is very challenging due to numerous contentious critical sections executed by a worker thread in the critical path of transaction execution [8]. State-of-the-art transaction processing systems achieve scalability by either 1) relying on partitioning, 2) using fine-grained synchronization that removes unbounded communication or 3) implementing specialized lock-free data structures. Many such approaches require radical redesigns of OLTP systems or are incompatible with one another.

Recently, Intel introduced a new processor microarchitecture, under a code name Haswell, that supports hardware transactional memory (HTM). It provides two ways of using HTM: 1) through specialized instructions that offer transactional primitives and 2) by using lock elision [16]. Specialized instructions allow a programmer to wrap a critical section inside a transaction, thus providing atomicity, consistency, and isolation. Lock elision is a technique aimed at improving the performance of existing lock based code. Threads speculatively execute the critical section inside a transaction without acquiring the lock. If their executions don't interfere with each other, they will commit and make their changes visible atomically to other threads. Otherwise, hardware will roll back and try to acquire the lock using non-transactional code paths.

[*] Work done while author was at EPFL.

DaMoN'15, May 31 - June 4, 2015, Melbourne, VIC, Australia
Copyright 2015 ACM 978-1-4503-2971-2/14/06 ...$15.00.
http://dx.doi.org/10.1145/2771937.2771946.

Numerous critical sections make OLTP applications an ideal target for applying HTM. In this paper, we quantify the challenges and opportunities for applying HTM to the complex transaction manager Shore-MT. Shore-MT uses a number of techniques to provide scalability on multicores using various synchronization mechanisms and techniques that limit contention on shared data structures. We replace different synchronization mechanisms with their HTM counterparts and attempt to entirely replace locks with hardware transactions for common database operations. While HTM can improve performance in some cases, the limitations of the current hardware implementation can also cause performance drops in case of frequent aborts. Effectively using HTM in existing complex system requires more drastic redesign of multiple components to work around the hardware limitations.

Our main contributions are:

- We replace various synchronization mechanisms with their HTM implementations in Shore-MT and show that this can improve throughput by 13-17% when running the TATP workload.
- We show that excessive use of hardware transactions can decrease performance due to aborts related to limitations of the current hardware implementation.
- We demonstrate that a straightforward encapsulation of a transactional operation (e.g., a B-tree probe) using HTM can severely hurt performance (up to 73%) due to the excessive length of the hardware transaction leading to long rollbacks upon aborts.

The rest of this paper is organized as follows: Section 2 presents the main features of Intel's hardware transactional memory implementation and related work on utilizing HTM. Section 3 provides experimental setup and an overview of our HTM implementation and discusses challenges we faced. We discuss experimental results in Sections 4 and 5. Finally, Section 6 concludes the paper and outlines future work.

2. BACKGROUND AND RELATED WORK

This section briefly introduces hardware transactional memory, discusses Intel's implementation, and surveys related work that utilizes hardware transactional memory to improve performance of software systems.

2.1 Hardware Transactional Memory

Transactional memory was introduced by Herlihy and Moss over 20 years ago [4]. In their seminal paper, they argue that lock-free data structures avoid common problems locking techniques exhibit, such as priority inversion, convoying and deadlocks, and that transactional memory makes lock-free synchronization as efficient as the lock-based one. They define a transaction as a sequence of instructions that is atomic and serializable, and argue that it can be implemented as a straightforward extension of the cache coherence protocol. Interestingly, two early commercial implementations use completely different implementations of hardware transactional memory compared to the original proposal. Sun's prototype Rock processor relies on speculative execution to implement best-effort HTM [1], while IBM's BlueGene/Q processor uses multiversioned last-level cache with unmodified cores and L1 caches for the same purpose [20].

Intel Transactional Synchronization Extension (TSX) is the new instruction set that appears in Intel's Haswell line of processors and enables transactional memory support in hardware. It is closer in spirit to the original HTM proposal than the previous commercial implementations. TSX instructions are implemented as an extension of the cache-coherency protocol, so they keep track of what memory addresses are accessed at a cache-line granularity. Current implementation is limited to the L1 data caches that are used to store both read and write sets of a transaction. The associativity of the cache (8 in current processors) as well as the size of the cache limits the size of these sets. An eviction of a write address from the cache always causes an abort. At the same time, a read address may be evicted from the cache before a transaction ends without causing an abort, due to limited support in cache coherence protocols for the private L2 caches.

TSX instructions can be used in two ways, through Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM) modes. Hardware Lock Elision is a legacy compatible API inspired by speculative lock elision (SLE) technique that improves performance of lock-based programs when critical sections could have been executed without locks [16]. TSX provides two instruction prefixes XACQUIRE and XRELEASE. The write to the lock prefixed with XACQUIRE is elided and the lock's address is added to the read-set. Further threads will see the lock as free and will be able to enter the critical section concurrently. On XRELEASE, the processor has to restore the lock to the value prior to XACQUIRE. If the value is unchanged, the processor will elide the write and won't conflict with the other threads' read sets. If no other execution has conflicted with a committing transaction, it will commit. Otherwise, the processor will roll back and explicitly acquire the lock.

Restricted transactional memory provides Haswell specific instructions XBEGIN, XEND, and XABORT. XBEGIN starts a transaction. Since starting a transaction can fail, the programmer needs to provide a fallback path. Also, the transaction's status code returned by XBEGIN allows a programmer to use a custom retry policy to address different reasons for a failure. XABORT aborts the transaction and will always succeed, while XEND tries to commit the transaction and also requires a fallback path. It does not provide any guarantees that commit will eventually succeed. Finally, both HLE and RTM have the XTEST instruction to test if its call is made within a transaction. XTEST is very useful since calling XRELEASE or XEND outside of a transaction will result in a very expensive interrupt.

2.2 Related work

Intel's Haswell processors have been used in a number of recent studies that aim at analyzing the performance of different types of software systems. Intel's study shows that TSX improves performance of high performance computing workloads by 40% on average and gives 30% performance improvement in network intensive applications when TSX is applied to the TCP/IP stack [24]. This improvement mainly comes from avoiding serialization and reducing the cost of uncontended critical sections. A recent study of applying TSX to garbage collectors demonstrates performance improvement of 1.5-2x, while achieving similar improvement with software transactional memory (STM), which suggests that hardware implementations have a lot of space for improvements [17]. Finally, a comparison study of a standard transaction memory suite as well as a number of other stress tests concludes that TSX outperforms software approaches by 3.3x for short critical sections, while both locking and STM are still better for long critical sections [2].

The roots of the idea of transactional memory can be traced to the early research on the concepts of database transactions [14]. More recently, a study has compared early hardware prototypes and simulation of HTM to conclude that HTM is attractive for low contention scenarios and can be combined with spinlocks to implement an efficient lock manager in a database system [19]. A number of projects have investigated the applicability of Intel's Haswell for designing high performance database systems. The HyPer team has proposed a very low overhead concurrency control mechanism that combines timestamp ordering with short hardware transactions [13]. A recent study has demonstrated that TSX can improve performance of operations on common tree index structures [10]. Finally, a recent proposal demonstrates that a design tuned for Intel's RTM instructions can offer performance comparable to a state-of-the-art main memory transaction processing system with fine-grained locks while having lower code complexity [21]. Our study is complementary to these results, since we use a complete complex transaction processing system as a starting point and evaluate the applicability of HTM without redesigning any of the components.

3. SETUP AND METHODOLOGY

Shore-MT is a scalable open source storage manager optimized for multicore processors [9]. It uses a number of different synchronization mechanisms tuned for a particular communication pattern in each component of the system. Worker threads in Shore-MT use fine-grained synchronization to avoid any scalability bottlenecks. Even though we don't observe any obvious scalability bottlenecks when running Shore-MT with 8 threads, it goes through around 70 critical sections even for a single row update transaction [8]. Minimizing the time spent in critical section has a potential for performance improvement. In this section, we outline the experimental setup and methododology and describe how we augment existing implementations of different locks with the HTM versions before presenting experimental analysis in the next two sections.

3.1 Experimental platform

Hardware and OS. In all experiments we use a server with Intel's Haswell i7-4770 processor running at 3.4Ghz. It has 16GB of main memory. We use memory mapped disks for storing data and logs since we don't have an I/O subsystem that can sustain the generated load, and we focus on maximally utilizing CPU while exercising all code paths in the system. The operating system used in all experiments is RedHat Enterprise Linux version 6.5 with kernel version 2.6.32 and we compile code using gcc version 4.8 with maximum optimizations.

Profiling tools. For performance analysis, we use version 2.6 of Intel's Performance Counter Monitor (PCM) tool [22]. PCM can track the number of started, committed, and aborted transactions and distinguish different causes of aborts such as data conflict, write buffer capacity, and unfriendly instructions. To access additional information (e.g., the location of an abort) we have used the new experimental features of Intel VTune Amplifier XE version 2015 [18].

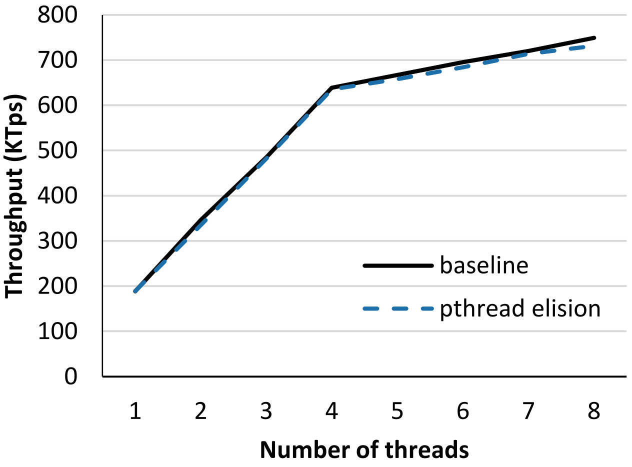

Figure 1: Impact of pthread lock elision on the throughput for TATP GetSubData.

Software. As a software platform, we use Shore-MT with enabled Speculative Lock Inheritance (SLI) [7]. SLI lets the worker threads inherit the locks that are accessed mostly in read mode from one transaction to another. This reduces the number of times a thread needs to go to the lock manager to request a lock because threads can keep these mostly read-mode locks they acquired in the previous transaction in the next one. It is especially useful for HTM since it reduces abort rates of hardware transactions.

Workload. We use the TATP benchmark in two flavors: only the GetSubscribersData (*GetSubData*) transaction or the whole TATP mix (*Mix*) [15]. TATP models operations of a mobile phone provider and has very short transactions that particularly stress thread synchronization of a transaction processing system. GetSubData is a very short transaction that reads only one row. We run experiments for 30 seconds and repeat each experiment three times to obtain results with standard deviation of less than 1%.

Runs. We use a dataset with 80000 subscribers (120MB) and increase the number of worker threads in the experiments from 1 to 8 to evaluate scalability. For runs with 1 to 4 threads we use one thread per core, while for more threads we also use hyper-threading.

3.2 HTM synchronization mechanisms

We first illustrate the use of HTM on occ_rwlock that is a read/write lock used in Shore-MT to protect histograms and ensure correctness during the checkpoint operation. It is accessed in read mode in the critical path of transaction execution. Since it can also be implemented using pthread mutexes or replaced by using pthread_rwlock, we first attempted to use these options. The pthread library provides experimental support for HLE [23, 11].

Figure 1 shows throughput with and without pthread-enabled lock elision for TATP's GetSubData transaction. We observe that enabling elision in pthread library has negligible impact on performance. The version we used relies on conditional variables that are causing transactions to abort. Additionally, according to Chapter 12 of Intel's optimization guide [6] regarding TSX, HLE implementation is more constrained than the RTM one. In particular, Haswell's implementation of HLE supports only single level nesting, compared to seven levels in RTM. Also, certain aborts can only happen when using HLE, e.g., unaligned or partially overlapping accesses to an elided lock variable cause an abort. Finally, RTM implementations can be much more flexible as programmers can fine tune the conflict resolution policies as we describe in the paragraphs below. Due to its higher flexibility, we use RTM in the rest of this paper.

Since using HTM support in the pthread library did not work well for this case, we augment our existing implementation of occ_rwlock using RTM. As any changes should transparently fall back to the old implementation on platforms that do not support TSX, we use preprocessor directives to isolate the RTM code. We follow the best practices described by Intel's guidelines [6, 12].

The interface of occ_rwlock consists of two acquire and two release functions: one of each kind for reads and writes. We extend implementation of the *acquire* functions in the following way:

```
void occ_rwlock::acquire() {
#ifdef OCC_RWLOCK_RTM_WRAPPER
  unsigned int status;
  for(int i = 0; i < 2; i++) {
    if ((status = _xbegin()) == _XBEGIN_STARTED) {
      if (has_reader()) {
        _xabort(0xff);
      }
      return;
  } else if ((status & _XABORT_EXPLICIT) &&
              _XABORT_CODE(status) == 0xff) {
       while (__atomic_load_n(&_active_count,
                             __ATOMIC_ACQUIRE))
        _mm_pause();
      } else if (status & _XABORT_CONFLICT) {
        long int backoff=10*random()/RAND_MAX;
        while (backoff--) _mm_pause();
      } else if (status & _XABORT_RETRY) {
        _mm_pause();
      } else {
        break;
      }
  }
#endif
  /**original acquire code here**/
}
```

We follow Intel's guidelines that suggest using 1-4 attempts to start a transaction before falling back to non-transactional path. Our experiments show that 2 attempts are optimal for our scenario. We make sure to avoid the lemming effect, i.e., the case where threads prevent each other from starting a TSX transaction by always acquiring the lock [12]. In case of an explicit abort (i.e., when the lock is held by another thread), the obvious solution is to wait until the lock is free again and retry the operation (i.e, the same behaviour as the normal lock).

However, if an abort is caused by other reasons, we can adapt and tune the fallback path. There may be several reasons an abort occurs, so we adapt our strategy according to the status code returned by _xbegin(). In particular, in case of data conflicts, we pause (__mm_pause()) for a random amount of time and retry later. If the abort is due to another reason (e.g., overflow of the transactional state buffer) and the operation might succeed next time, we can retry it immediately. Otherwise, we acquire the lock. This implementation gives priority to TSX transactions over acquiring the lock.

The modifications to the *release* functions are more straightforward:

```
void occ_rwlock::release() {
#ifdef OCC_RWLOCK_RTM_WRAPPER
  if (!has_reader() & _xtest()) {
    _xend();
    return;
  }
#endif
  /**original release code here**/
}
```

We use the same approach to elide other locks. The two spinlocks that we elide are tatas (test-and-test-and-set) and mcslock. Tatas lock is a fast spinlock used by Shore-MT for its lightly contended critical sections (e.g., latching, logging), as it does not scale well under heavy contention.

Mcslock is a queue-based spinlock useful for short, contented critical sections. To acquire the lock, a thread has to atomically override the queue's tail with its pointer. If the previous value stored in the tail isn't NULL, the thread spins on it until it becomes NULL again. The tail variable constitutes the lock in itself. In addition to *acquire* and *release* functions common to previously discussed locks, the mcslock also has an *attempt* function which only acquires the lock if it is free. Extending the attempt function is simpler compared to the *acquire* function and is illustrated in the listing below:

```
bool attempt(qnode* me) {
#ifdef MCS_LOCK_RTM_WRAPPER
  unsigned int status;
  if ((status = _xbegin()) == _XBEGIN_STARTED) {
    if (_tail != NULL) {
      _xabort(0xff);
    }
    return true;
  }
#endif
  /**originial code here**/
}
```

The final lock that we examine is single reader, multiple writer srwlock that is used for latching in Shore-MT. Its implementation relies on mcslocks, so we could reuse RTM-enabled mcslocks with small modifications related to the complex interface of srwlock. This complexity arises since srwlock should be able to return the number of readers and the mode in which it is held.

4. APPLYING HTM TO LOCKS

In this section, we analyze the impact of the modifications presented in Section 3.2. We identify the cases in which HTM improves performance and the cases when it harms performance, and analyze the causes for such effects.

4.1 Beneficial cases

We start with the best case, when eliding locks improves performance. We use RTM-enabled versions of occ_rwlock, tatas lock and mcslock, while keeping the vanilla srwlock,

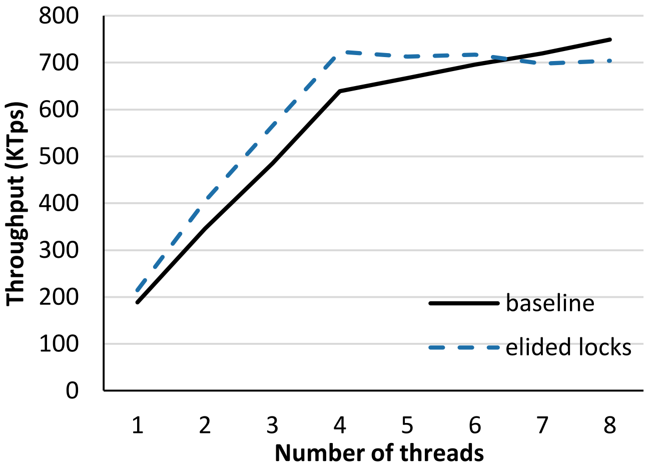

Figure 2: Impact of eliding selected locks (occ_rwlock, tatas lock, mcslock) for TATP GetSubData.

threads	% aborts	% capacity	% conflict
1	3.5	2.7	0.7
2	3.2	1.5	1.7
3	5.2	1.2	4.0
4	7.2	0.6	6.8
5	11.1	2.3	7.3
6	12.3	3.2	6.0
7	14.3	5.2	5.2
8	16.1	6.0	5.0

Table 1: Breakdown of abort rates by root cause for different number of worker threads when eliding selected locks (occ_rwlock, tatas lock, mcslock).

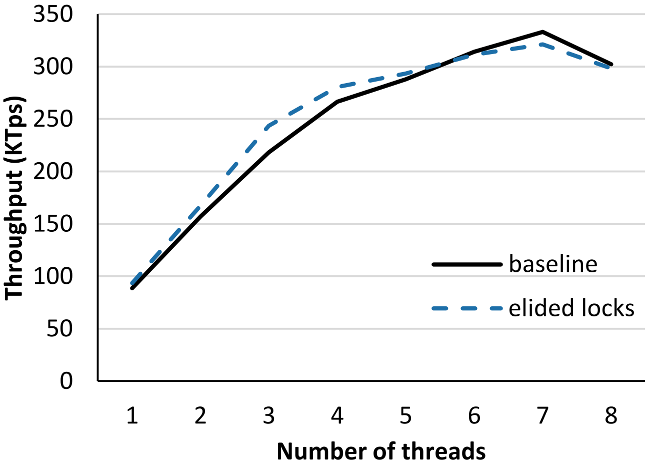

Figure 3: Impact of eliding selected locks (occ_rwlock, tatas lock, mcslock) for TATP Mix workload.

and run TATP GetSubData and Mix. Figure 2 and Figure 3 plot the throughput of the baseline and the version that uses RTM-enabled synchronization primitives.

We get the best improvement when running up to one thread per core for read-only GetSubData transactions, ranging from 13% to 17%. The improvement is smaller in the case of TATP Mix workload due to the larger number of critical sections and the mix of read-only and update transactions. Executing more critical sections causes more aborts since it increases the potential of collisions and conflicts.

Beyond 4 threads, for GetSubData, we observe that the increase in throughput decays with elided locks whereas the baseline throughput keeps increasing at a higher rate. To shed some light on this behavior, we use the PCM tool. Since it can measure up to 4 hardware counters per run, we opt for measuring the number of hardware transactions started and aborted, as well as the number of hardware transactions that have aborted via capacity and conflict signal. Capacity signal is triggered when there are is no available space in the transactional store buffer which is significantly smaller than the size of the L1 data cache. Conflict is signaled due to a data conflict on a transactionally accessed address. All other aborts are tracked using the miscellaneous hardware counter. We profile the GetSubData and report the computed total abort rates as well as the capacity and conflict rates in Table 1. As single abort can trigger multiple signals, the sum of capacity and conflict rates can be higher than the overall abort rate [6].

The abort rate is fairly low until 4 threads: around 3% for 1 and 2 threads and below 10% for 3 and 4 threads. In all our experiments, we observe that capacity conflicts are the main cause of aborts for single threaded case whereas their contribution diminishes with more threads. With 2 to 4 threads, conflicts become the main cause of aborts which is expected due to the concurrent accesses to the protected data structures. When using hyper-threading, abort rates jump over 10% and keep increasing with more threads. Capacity constraints become a significant contributor to the abort rates, as well as resource limitations caused by two threads sharing the same L1 cache. The Mix workload observes a similar behavior.

4.2 Challenging case

Not all lock elision is beneficial, as we show in this experiment where we use the RTM versions of all four locks. We run GetSubData that benefits from using RTM-enabled versions of occ_rwlock, tatas lock and mcslock, and we additionally use RTM version of srwlock. Figure 4 plots the throughput as the number of threads increases while Table 2 contains a breakdown of aborts. The version with elision is 16% slower on average between 2 and 5 threads due to the abort rates of over 18%. Conflicts are the main contributor to the high abort rates. In our system these are real data conflicts on statistic variables accessed in the critical sections protected by srwlock. Resolving this issue requires modifications to the system to remove these data conflicts, for example, by decentralizing statistics collection. The relative impact of different causes follows the trend we observed in Table 1. The impact of capacity aborts is high in the single threaded case, then diminishes for 2-4 threads and again rises steadily when there are multiple thread per core. Resource aborts are also more prominent in the presence of hyper-threading.

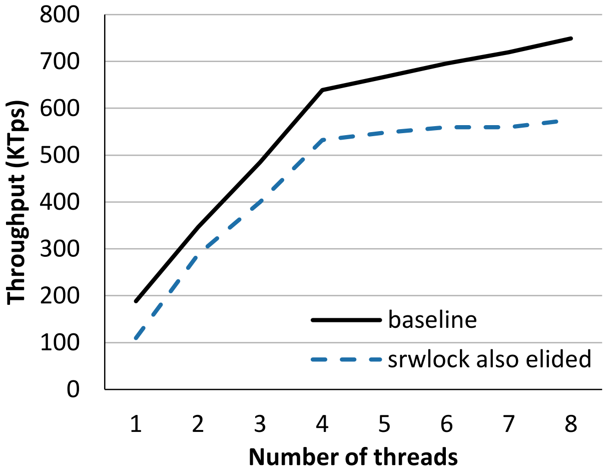

Figure 4: Impact of eliding all locks (occ_rwlock, tatas lock, mcslock, srwlock) for TATP GetSubData.

threads	% aborts	% capacity	% conflict
1	15.2	13.8	1.3
2	18.2	2.2	17.9
3	19.8	1.9	20.8
4	20.4	2.8	22.1
5	20.9	3.0	18.5
6	22.2	4.5	15.8
7	23.1	5.3	13.3
8	25.1	6.1	12.1

Table 2: Breakdown of abort rates by root cause for different number of worker threads when eliding all locks (occ_rwlock, tatas lock, mcslock, srwlock).

Discussion. As we demonstrate in this section, it is possible to achieve performance improvements by using HTM to implement synchronization mechanisms used in a complex application with fine-grained locking. While the improvement is possible even when running in single threaded mode, using HTM can also decrease performance when transactions have high abort rates. Decreasing abort rates requires more detailed root cause analysis to remove lurking contention on shared data structures and careful padding to eliminate any aborts due to false sharing.

5. APPLYING HTM TO B-TREE OPERATIONS

In order to evaluate the applicability of HTM on more complex operations in the database system, we replace fine-grained latching during B-tree traversal with a single coarse-grained latch. We modify the tree traversal function by placing an mcslock (that will be elided) to protect the call to the lookup function. In this way, threads will just elide one lock instead of acquiring/releasing latches as they traverse the B-tree to find the key. This approach exemplifies the spirit of transactional memory: use a coarse-grained lock on a data structure to simplify programming and let transactional memory exploit the inner parallelism.

We evaluate the impact of eliding locks in B-tree traversal code using GetSubData workload and plot the performance in Figure 5. The results are disappointing: the throughput is 73% lower than the baseline with 8 threads. Even for a single thread, the throughput is 10% lower. We profile the runs using the PCM tool [22] and summarize the abort rates in Table 3 breaking them down into capacity and conflict categories. The abort rates for 2 to 4 threads average 10% before steadily climbing when we also use hyper-threading. While the percentage of capacity related aborts is roughly similar to the case where using RTM was beneficial, the percentages of conflict and other resource related aborts rise. Also, the impact of these aborts is much higher due to very long transactions that cause contention and waste significant amount of work done when they're aborted. Hence, HTM is much more suitable for short critical sections.

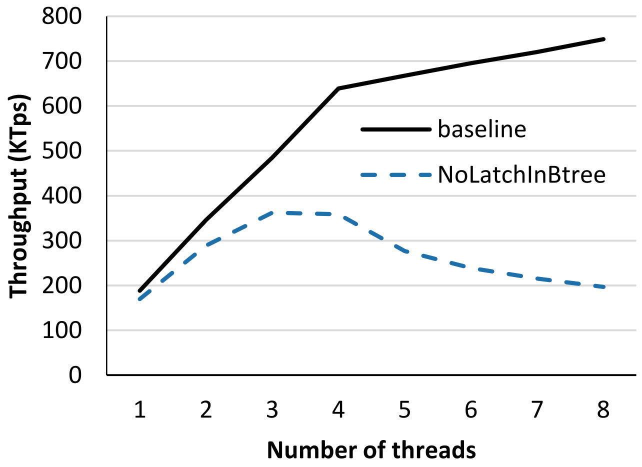

Figure 5: Impact of lock elision in B-tree traversal code for TATP GetSubData.

threads	% aborts	% capacity	% conflict
1	6.1	5.1	0.9
2	10.5	1.4	2.3
3	10.8	0.7	6.0
4	10.7	0.7	4.2
5	12.8	1.1	4.3
6	14.3	1.8	4.8
7	15.2	2.0	4.2
8	15.5	2.1	4.3

Table 3: Breakdown of abort rates by their root cause when eliding a coarse grained lock for different number of worker threads.

Discussion. At first glance, these results contradict the conclusion of the study of using HTM to improve performance of index structures for main memory databases [10]. However, to achieve good performance, authors of the study had to modify data structures to overcome limitations of the current hardware implementation. Also, they are studying the indexes in isolation whereas we are looking at the whole system which increases 1) the size of the hardware transactions since we need to ensure isolation and 2) the probability of aborts due to capacity limitations of the L1 cache. Achieving comparable results in our system is possible, however it requires holistic redesign of multiple components of the system.

6. CONCLUSIONS AND FUTURE WORK

In this paper we present a study of directly applying Intel's TSX instructions to Shore-MT transaction manager. We analyze two scenarios: using RTM to optimize implementation of the synchronization mechanisms and using RTM-enabled lock to replace multiple locks in a B-tree traversal operation. The first scenario shows promising results since TSX is very efficient for short critical sections with low contention. On the other hand, applying it to complex operations requires the use of long critical sections that cause expensive aborts due to the limitations of the current implementation.

In the months after the release of Haswell desktop processors like the one we use in this study, Intel has found a bug in the TSX implementation and disabled TSX support in affected processors [5]. While this illustrates challenges in designing hardware transactional memory, we are optimistic that it shows great potential for improving the performance of multithreaded software. Shore-MT is an example of a scalable and complex system whose performance can be improved using TSX on a benchmark that stresses synchronization facilities. However, utilizing the full potential of transactional memory on current and future hardware platforms requires a substantial redesign of different components of the system.

Acknowledgements

We would like to thank the members of the DIAS laboratory for their support throughout this work. We also thank the reviewers for their constructive feedback. This work is partially funded by the Swiss National Science Foundation (Grant No. 200021-146407/1).

7. REFERENCES

[1] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *ASPLOS*, pages 157–168, 2009.

[2] N. Diegues, P. Romano, and L. Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. In *PACT*, pages 3–14, 2014.

[3] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM Can Be More Than a Research Toy. *CACM*, 54(4):70–77, 2011.

[4] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[5] Intel. Erratum HSW136: Software Using Intel TSX May Result in Unpredictable System Behavior. http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf.

[6] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[7] R. Johnson, I. Pandis, and A. Ailamaki. Improving OLTP scalability using speculative lock inheritance. *PVLDB*, 2(1):479–489, 2009.

[8] R. Johnson, I. Pandis, and A. Ailamaki. Eliminating unscalable communication in transaction processing. *VLDBJ*, 23(1):1–23, 2014.

[9] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009.

[10] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with Intel Transactional Synchronization Extensions. In *HPCA*, pages 476–487, 2014.

[11] A. Kleen. https://github.com/andikleen/glibc.

[12] A. Kleen. TSX anti patterns in lock elision code. https://software.intel.com/en-us/articles/tsx-anti-patterns-in-lock-elision-code.

[13] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE*, pages 580–591, 2014.

[14] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *LDRS*, pages 128–137, 1977.

[15] S. Neuvonen, A. Wolski, M. Manner, and V. Raatikka. Telecom application transaction processing benchmark (TATP), 2009. http://tatpbenchmark.sourceforge.net/.

[16] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO*, pages 294–305, 2001.

[17] C. G. Ritson, T. Ugawa, and R. E. Jones. Exploring Garbage Collection with Haswell Hardware Transactional Memory. In *ISMM*, pages 105–115, 2014.

[18] K. Rogozhin. Profiling Intel Transactional Synchronization Extensions with Intel VTune Amplifier XE. https://software.intel.com/en-us/articles/profiling-intel-transactional-synchronization-extensions-with-intel-vtune-amplifier-xe.

[19] K. Q. Tran, S. Blanas, and J. F. Naughton. On transactional memory, spinlocks, and database transactions. In *ADMS*, pages 43–50, 2010.

[20] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *PACT*, pages 127–136, 2012.

[21] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Eurosys*, pages 26:1–26:15, 2014.

[22] T. Willhalm, R. Dementiev, and P. Fay. Intel performance counter monitor 2.6, 2012. http://www.intel.com/software/pcm.

[23] N. Willis. Merging lock elision into glibc. https://lwn.net/Articles/557222/.

[24] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *SC*, pages 1–11, 2013.