2014 IEEE 28th International Parallel & Distributed Processing Symposium

Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell

Bhavishya Goel, Ruben Titos-Gil, Anurag Negi, Sally A. McKee, Per Stenstrom
Chalmers University of Technology
Gothenburg, Sweden
{*goelb, ruben.titos, negi, mckee, per.stenstrom*}*@chalmers.se*

***Abstract*—Hardware transactional memory implementations are becoming increasingly available. For instance, the Intel Core™ i7 4770 implements Restricted Transactional Memory (RTM) support for Intel Transactional Synchronization Extensions (TSX). In this paper, we present a detailed evaluation of RTM performance and energy expenditure. We compare RTM behavior to that of the TinySTM software transactional memory system, first by running microbenchmarks, and then by running the STAMP benchmark suite. We find that which system performs better depends heavily on the workload characteristics. We then conduct a case study of two STAMP applications to assess the impact of programming style on RTM performance and to investigate what kinds of software optimizations can help overcome RTM's hardware limitations.**

I. Introduction

Transactional memory (TM) [1] simplifies some of the challenges of shared-memory programming. The responsibility for maintaining mutual exclusion over arbitrary sets of shared-memory locations is devolved to the TM system, which may be implemented in software (STM) or hardware (HTM). TM presents the programmer with fairly easy-to-use programming constructs that define a *transaction* — a piece of code whose execution is guaranteed to appear as if it occurred atomically and in isolation.

The research community has explored this design space in depth, and a variety of proposed systems take advantage of transaction characteristics to simplify implementation and improve performance [2]–[5]. Hardware support for transactional memory has been implemented in Rock [6] from Sun Microsystems, Vega from Azul Systems [7], and Blue Gene/Q [8] and System z [9] from IBM. Haswell is the first Intel product to provide such hardware support. Intel's Transactional Synchronization Extensions (TSX) allow programmers to run transactions on a best-effort HTM implementation, i.e., the platform provides no guarantees that hardware transactions will commit successfully, and thus the programmer must provide a non-transactional path as a fallback mechanism. Intel TSX supports two software interfaces to execute atomic blocks: Hardware Lock Elision (HLE) is an instruction set extension to run atomic blocks on legacy hardware, and Restricted Transactional Memory (RTM) is a new instruction set interface to execute transactions on the underlying TSX hardware.

Here we compare the Haswell RTM performance and energy of the Haswell implementation of RTM to those of other approaches for controlling concurrency. We use a variety of workloads to test the susceptibility of RTM's best-effort nature to performance degradation and increased energy consumption. We compare RTM performance to TinySTM, a software transactional memory implementation that uses time to reason about the consistency of transactional data and about the order of transaction commits.[1] We highlight these crossover points and analyze the impact of thread scaling on energy expenditure.

We find that RTM performs well with small to medium working sets when the amount of data (particularly that being written) accessed in transactions is small. When data contention among concurrent transactions is low, TinySTM performs better than RTM, but as contention increases, RTM consistently wins. RTM generally suffers less overhead than TinySTM for single-threaded runs, and it is more energy-efficient when working sets fit in cache.

II. Experimental Setup

The Intel 4th Generation Core™ i7 4770 processor comprises four physical cores that can run up to eight simultaneous threads when hyper-threading is enabled. Each core has two eight-way 32 KB private L1 caches (separate for I and D), a 256 KB private L2 cache (for combined I and D), and an 8 MB shared L3 cache, with 16 GB of physical memory on board. We compile all microbenchmarks, benchmarks, and synchronization libraries using gcc v4.8.1 with *-O3* optimization flag. We use the *-mrtm* flag to access the Intel TSX intrinsics. We schedule threads on separate physical cores (unless running more than four threads) and fix the CPU affinity to prevent migration.

We modify the *task* example from *libpfm4.4* to read both the performance counters and the processor package energy via the Running Average Power Limit (RAPL) [11] interface. We verify these energy figures against measurements at the ATX CPU power supply input of the motherboard and find them to be strongly correlated (Spearman's correlation

[1] We choose TinySTM because during our experiments we find that it consistently outperforms other STM alternatives like TL2 (to which RTM was compared in another recent study [10]).

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.70

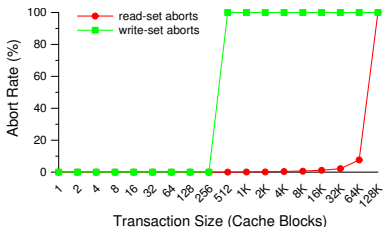


Figure 1. RTM Read-Set and Write-Set Capacity Test

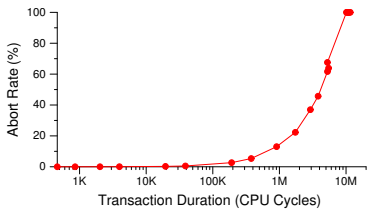


Figure 2. RTM Abort Rate versus Transaction Duration

coefficient ρ=0.9858). We implement Intel TSX synchronization as a separate library and add RTM definitions to the STAMP *tm.h* file. When transactions fail more than eight times, we invoke reader/writer lock-based fallback code to ensure forward progress. If the return status bits indicate that an abort was due to another thread's having acquired the lock (in the fallback code), we wait for the lock to be free before retrying the transaction. The following shows pseudocode for a sample transaction.

Algorithm 1 Implementation of `BeginTransaction`

```
while true do
    nretries ← nretries + 1
    status ← _xbegin()
    if status = _XBEGIN_STARTED then
        if arch_read_can_lock(serialLock) then
            return
        else
            _xabort(0)
        end if
    end if
    {*** fall-back path ***}
    while not arch_read_can_lock(serialLock) do
        _mmpause()
    end while
    if nretries ≥ MAX_RETRIES then
        break
    end if
end while
arch_write_lock(serialLock);
return
```

III. Microbenchmark Analysis

A. Basic RTM Evaluation

We first quantify RTM's hardware limitations that affect its performance using microbenchmark studies. We detail the results of these experiments below.

RTM Capacity Test. To test the limitations of read-set and write-set capacity for RTM, we create a custom microbenchmark, results for which are shown in Fig. 1. The abort rate of write-only transactions tops out at 512 cache blocks (the size of L1 data cache). We suspect this is because write-sets are tracked only in L1, and so evicting any transactionally written cache line from L1 results in a transaction abort. For read-sets, the abort rate saturates at 128K cache blocks (the size of L3 cache). This suggests that evicting transactionally read cache lines from L3 (but not L1) triggers transaction aborts, and thus RTM maintains performance for much larger read-sets than write-sets.

RTM Duration Test. Since RTM aborts can be caused by system events like interrupts and context switches, we study the effects of transaction duration (measured in CPU cycles) on success rate. For this analysis, we use a single thread, set the working-set size to 64 bytes, and set the number of writes inside the transaction to 0. This tries to ensure that the number of aborts due to memory events and conflicts remains insignificant. We gradually increase the duration by increasing the number of reads within the transaction. Fig. 2 shows that transaction duration begins to affect the abort rate at about 30K cycles and that durations of more than 10M cause all transactions to abort (note that these results are likely machine dependent).

RTM Overhead Test. Next we quantify performance overheads for RTM compared to spin locks and the atomic compare-and-swap (CAS) instruction. For this test, we create a microbenchmark that removes elements from a queue (defined in the STAMP [12] library). We initialize the queue to 1M elements, and threads extract elements until the queue is empty. Work is not statically divided among threads. We first compare RTM against the spinlock implementation in the Linux kernel (`arch/x86/include/asm/spinlock.h`). We then compare against a version of the queue implementation modified to use CAS in `queue_pop()`. For RTM, we simply retry the transaction on aborts.

We perform three sets of experiments. To observe the cost of starting an RTM transaction in the absence of contention, we first run single-threaded experiments. We repeat the experiment with four threads to generate a high-contention workload. Finally, we lower contention by making threads work on local data for a fixed number of operations after each critical section. Table I summarizes execution times normalized to those of the lock-based version.

Contention	Type of synchronization			
	None	Lock	CAS	RTM
None	0.64	1	1.05	1.45
Low	N/A	1	0.64	0.69
High	N/A	1	0.64	0.47

Table I
Relative Overhead of RTM versus Locks and CAS

Table I shows that the cost of starting a transaction makes RTM perform worse than the other alternatives when executing non-contended critical sections with few instructions. RTM suffers about a 45% slowdown compared to

Characteristic	Definition
Concurrency	Number of concurrently running threads
Working-set size[a]	Size of frequently used memory
Transaction length	Number of memory accesses per transaction
Pollution	Fraction of writes to total memory accesses inside transaction
Temporal locality	Probability of repeated address inside transaction
Contention	Probability of transaction conflict
Predominance	Fraction of transactional cycles to total application cycles

[a]Working-set size for Eigenbench is defined per-thread.

Table II
EIGENBENCH TM CHARACTERISTICS

using locks and CAS, and it takes over twice the time of an unsynchronized version. In contrast, our multi-threaded experiments reveal that RTM exhibits roughly 30% and 50% lower overhead than locks in low and high contention, respectively, while CAS is in both cases around 35% better than locks. Note that transactions avoid hold-and-wait behavior, which seems to give RTM an advantage in our study. When comparing locks and CAS, the higher lock overhead is likely due in part to the ping-pong coherence behavior of the cache line containing the lock and to cache-to-cache transfers of the line holding the queue head.

B. Eigenbench Characterization

To compare RTM and STM in detail, we next study the behaviors of Hong et al.'s Eigenbench [13]. This parameterizable microbenchmark attempts to characterize the design space of TM systems by orthogonally exploring different transactional application behaviors. Table II defines the seven characteristics we use to compare performance and energy expenditure of the Haswell RTM implementation and the TinySTM [14] software transactional memory system. Hong et al. [13] provide a detailed explanation of these characteristics and the equations used to quantify them.

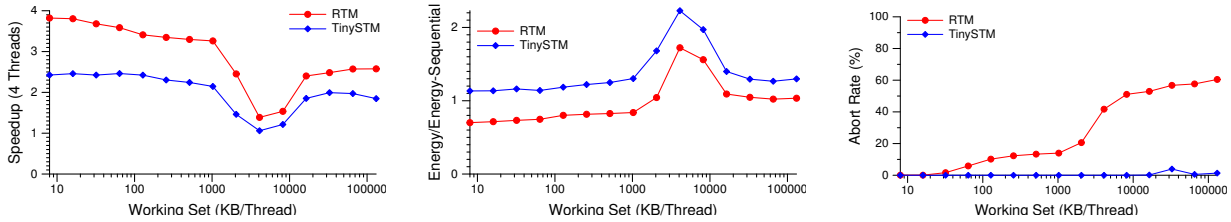
Unless otherwise specified, we use the following parameters in our experiments, results for which we average over 10 runs. Transactions are 100 memory references (90 reads and 10 writes) in length. We use one small (16KB) and one medium (256KB) working set size to demonstrate the differences in RTM performance. Since L1 size has no influence on TinySTM's abort rates, we only show TinySTM results for the smaller working set size. To prevent L1 cache interference, we run four threads with hyper-threading disabled as our default, and we fix the CPU affinity to prevent thread migration. For each characteristic, we compare RTM and TinySTM performance and energy (versus sequential runs of the same code) and transaction-abort rates. For the graphs in which we plot two working-set sizes for RTM, the speedups and energy efficiency given are relative to the sequential run of the same size working set.

Working-Set Size. Fig. 3 shows Eigenbench results over a logarithmic scale as we increase each thread's working set from 8KB to 128MB. RTM performs best with the smallest working set, and its performance gradually degrades as working-set size increases. The performance of both RTM and TinySTM drops once the combined working sets of all threads exceed the 8MB L3 cache. RTM performance suffers more because events like L3 evictions, page faults, and interrupts trigger a transaction abort, which is not the case for TinySTM. The speedups of both RTM and TinySTM are lowest at working sets of 4MB: at this point, the parallelized code's working sets (16MB in total) exceed L3, but the working set of the sequential version (4MB) still fits. For working sets above 4MB, the sequential version starts encountering L3 misses, and thus the relative performances of both transactional memory implementations begins to improve. Parallelizing the transactional code using RTM is energy-efficient compared to sequential version when the combined working sets of all threads fits inside the cache.

Transaction Length. Fig. 4 shows Eigenbench results as we increase the transaction length from 10 to 520 memory operations. When the working set (16KB) fits within L1, RTM outperforms TinySTM for all transaction lengths. For 256KB working sets, RTM performance drops sharply when the transaction length exceeds 100 accesses. Recall that evicting write-set data from the L1 triggers transaction aborts, but when the working set fits within L1, such evictions are few. As the working set grows, the randomly chosen addresses accessed inside the transactions have a higher probability of occupying more L1 cache blocks, and hence they are more likely to be evicted. In contrast, TinySTM shows no performance dependence on working-set size. The overhead of starting the hardware transaction affects RTM performance for very small transactions. As observed in the working-set analysis above, RTM is more energy efficient than both the sequential run and TinySTM for all transaction lengths when using the smaller working set. When using the larger working set, RTM expends more energy for transactions exceeding 120 accesses.

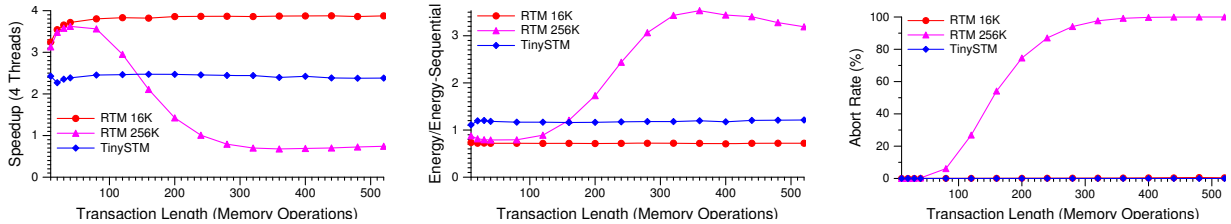
Pollution. Fig. 5 shows results when we test symmetry (with respect to handling read-sets and write-sets) by gradually increasing the fraction of writes. The pollution level is zero when all memory operations in the transaction are reads and one when all are writes. When the working set fits within L1, RTM shows almost no asymmetry. But for the larger working-set size, RTM speedup suffers as the level of pollution increases. TinySTM outperforms RTM when the pollution level increases beyond 0.4.

Temporal Locality. We next study the effects of temporal locality on TM performance (where temporal locality is defined as the probability of repeatedly accessing the same memory address within a transaction). The results in Fig. 6 reveal that RTM shows no dependence on temporal locality for the 16KB working set, but performance degrades for the



(a) Speedup (b) Relative Energy (c) Abort Rate

Figure 3. Eigenbench Working-Set Size



(a) Speedup (b) Relative Energy (c) Abort Rate

Figure 4. Eigenbench Transaction Length

256KB working set (where low temporal locality increases the number of aborts due to L1 write-set evictions). In contrast, TinySTM performance degrades as temporal locality increases, indicating that it favors unique addresses unless only one address is being accessed inside the transaction (locality = 1.0).

Contention. This analysis studies the behavior of TM systems when the level of contention is varied from low to high. We set the working-set size to 2MB for both RTM and TinySTM. The level of contention is calculated as an approximate value representing the probability of a transaction causing a conflict (as per the probability formula given by Hong et al. [13]). The conflict probability figures shown in Fig. 7 are calculated at word granularity and hence are specific to TinySTM. Since RTM detects conflicts at the granularity of cache line (64 bytes), the contention level is actually higher for RTM with the same workload configuration. When the degree of contention among competing threads is very low, RTM performs better than TinySTM. For low to medium contention, TinySTM considerably outperforms RTM. However, for high contention workloads, TinySTM performance degrades while RTM performance remains almost the same.

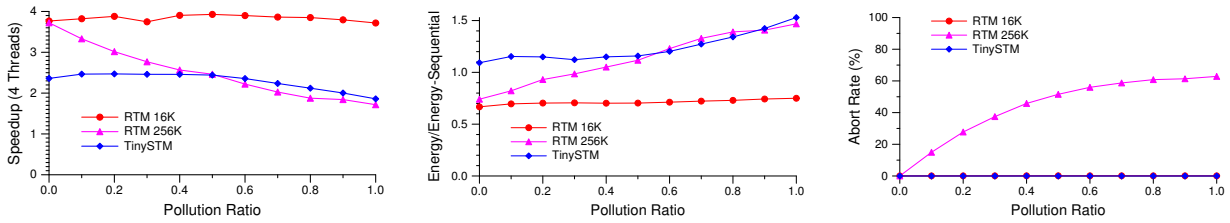
Predominance. We study the behavior of the TM systems when varying the fraction of application cycles executed within transactions to the total number of application cycles. For this analysis, we set working-set size to 256KB for both TM systems, we set contention to zero, and we vary the predominance ratio from 0.125 to 0.875. Fig. 8 shows that performance for both RTM and TinySTM suffers as the ratio of transactional cycles to non-transactional cycles grows. This can be attributed to the overheads associated with the TM systems: for the same level of predominance, TinySTM introduces more overhead because it must instrument the program memory accesses.

Concurrency. Next we study how the performance and energy of RTM and TinySTM scale when concurrency is increased from one thread to eight. Fig. 9 shows that RTM scales well up to four threads. At eight threads, the L1 cache is shared between two threads running on the same core. This cache sharing degrades performance for the larger working set more than for the smaller working set because hyper-threading effectively halves the write-set capacity of RTM. In contrast, TinySTM scales well up to eight threads. For the small working set, RTM proves to be more energy-efficient than either TinySTM or the sequential runs.

The results from the Eigenbench analysis help us in identifying a range of workload characteristics for which either RTM or TinySTM is better performing or more energy efficient. We next apply the insights gained from our microbenchmark studies to analyze the performance and energy numbers we see for the STAMP benchmark suite.

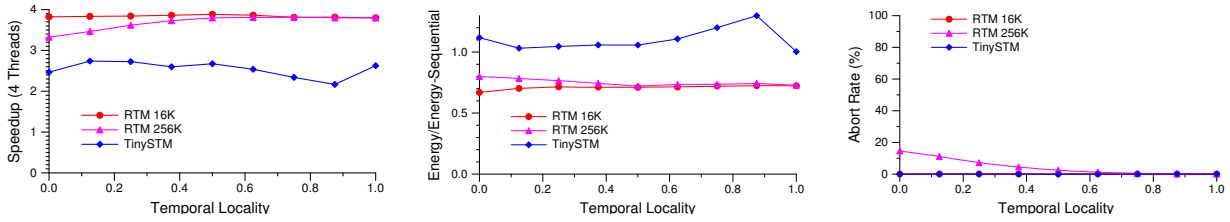
IV. HTM VERSUS STM USING STAMP

Next we use the STAMP transactional memory benchmark suite [12] to compare the performance and energy efficiency of RTM and TinySTM. We use the lock-based fallback mechanism explained in Section II and run the applications with input sizes that create large working sets and high contention. We average all results over 10 runs. Fig. 10 shows STAMP execution times for RTM and TinySTM normalized to the average execution time of sequential



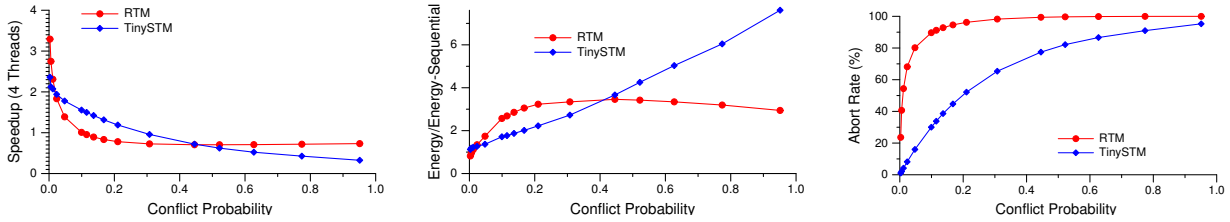
(a) Speedup (b) Relative Energy (c) Abort Rate

Figure 5. Eigenbench Pollution



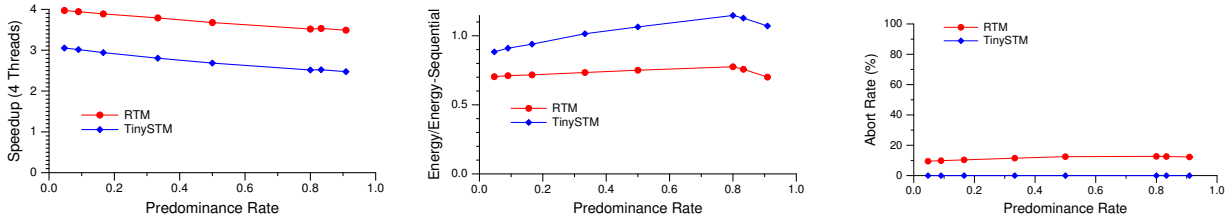
(a) Speedup (b) Relative Energy (c) Abort Rate

Figure 6. Eigenbench Temporal Locality



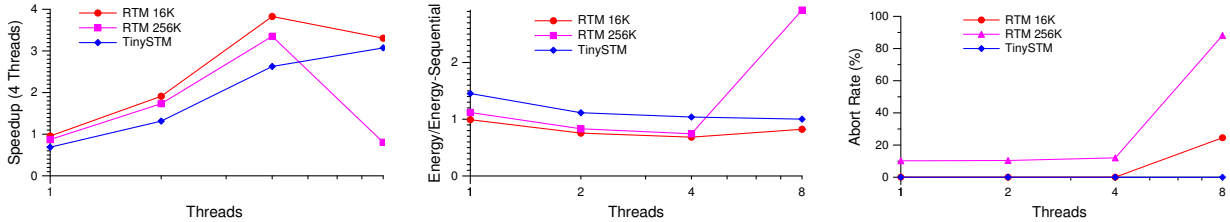
(a) Speedup (b) Relative Energy (c) Abort Rate

Figure 7. Eigenbench Contention



(a) Speedup (b) Relative Energy (c) Abort Rate

Figure 8. Eigenbench Predominance



(a) Speedup (b) Relative Energy (c) Abort Rate

Figure 9. Eigenbench Concurrency

(non-TM) runs. Fig. 11 shows the corresponding energy expenditures, again normalized to the average energy of the sequential runs. Results for single-threaded TM versions of the benchmarks illustrate the TM system overheads.

`bayes` has a large working set and long transactions, and thus RTM performs worse than TinySTM. This corresponds to our findings in the Eigenbench transaction-length analysis in Fig. 4. As expected, RTM does not improve the performance of `bayes` as the number of threads scales, and TinySTM performs better overall. Since the time the `bayes`'s algorithm takes to learn the network dependencies depends on the computation order, we see significant deviations in learning times for multi-threaded runs.

`genome` has medium transactions, a medium working-set size, and low contention. Most transactions have fewer than 100 accesses. Recall that in the working-set analysis shown in Fig. 3(a) (for transaction length 100), RTM slightly outperforms TinySTM for working-set sizes up to 4MB. On the other hand, TinySTM outperforms RTM when contention is low (Fig. 7(a)). The confluence of these two factors within `genome` yields similar performances for RTM and TinySTM up to four threads. For eight threads, as expected, TinySTM's performance continues to improve, whereas RTM's suffers from increased resource sharing among hyper-threads.

`intruder` is also a high-contention benchmark. As with `genome`, RTM performance scales well from one to four threads. Since `intruder` executes very short transactions, scaling to eight threads does not cause as much resource contention as for `genome`, and thus RTM and TinySTM perform similarly. Even though this application has a small to medium working set — which might otherwise give RTM an advantage — its performance is dominated by very short transaction lengths.

`kmeans` is a clustering algorithm that groups data items in N-dimensional space into K clusters. As with `bayes`, our 10 runtimes deviate significantly for the multi-threaded versions. On average, RTM performs better than TinySTM. The short transactions experience low contention, and the small working set has high locality, all of which give RTM a performance advantage over TinySTM. Even though both TM systems show speedups over the sequential runs, synchronizing the `kmeans` algorithm in TinySTM expends more energy at all thread counts.

`labyrinth` routes a path in a three-dimensional maze, where each thread grabs a start and an end point and connects them through adjacent grid points. Fig. 10 shows that labyrinth does not scale in RTM. This is because each thread makes a copy of the global grid inside the transaction, triggering capacity aborts that eventually cause the fallback to using a lock. Energy expenditure increases for the RTM multi-threaded runs because the threads try to execute the transaction in parallel but eventually fail, wasting many instructions while increasing cache and bus activity.

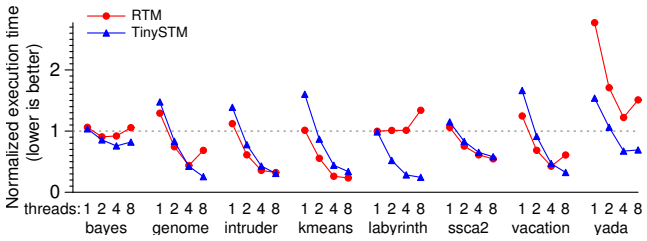


Figure 10. RTM versus TinySTM Performance for STAMP Benchmarks

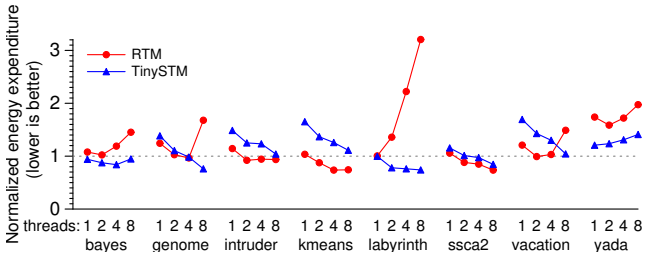


Figure 11. RTM versus TinySTM Energy Expenditure for STAMP Benchmarks

`ssca2` has short transactions, a small read-write set, and low contention, and thus even though it has a large working set, it scales well to higher thread counts. Performance for eight threads is good for both RTM and TinySTM. In general, RTM performs better (with respect to both execution time and energy expenditure) but not by much, as is to be expected for very short transactions.

`vacation` has low to medium contention among threads and a medium working set size. The transactions are of medium length, locality is medium, and contention is low. Like `genome`, `vacation` scales well up to four threads, but performance degrades for eight threads because its read-write set size is large enough that cache sharing causes resource limitation issues.

`yada` has big working set, medium transaction length, large read-write set, and medium contention. All these conditions give TinySTM a consistent performance advantage over RTM at all thread counts.

Our results in Fig. 11 indicate that the energy trends of applications do not always follow their performance trends. Applications like `bayes`, `labyrinth`, and `yada` expend more energy as they are scaled up, even when performance changes little (or even improves, in the case of `yada`). Only `intruder`, `kmeans`, and `ssca2` benefit from hyper-threading under RTM. In contrast, most STAMP applications benefit from hyper-threading under TinySTM, and those that do not suffer only small degradations.

Fig. 12 shows the overall abort rates for all benchmarks, including the contributions of different abort types. Based on our observations of hardware counter values, the current

Abort Type	Description
Data-conflict/ Read-capacity	Conflict aborts and read-set capacity aborts
Write-capacity	Write-set capacity aborts
Lock	Conflict and explicit aborts caused by serialization locks
Misc3	Unsupported instruction aborts[a]
Misc5	Aborts due to none of the previous categories[b]

[a] includes explicit aborts and aborts due to page fault/page table modification
[b] interrupts, etc.

Table III
INTEL RTM ABORT TYPES

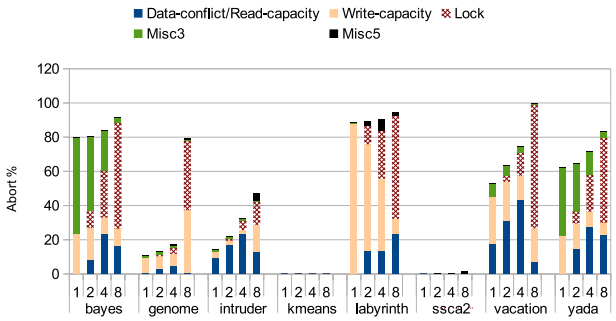


Figure 12. RTM Abort Distributions for STAMP Benchmarks

RTM implementation does not seem to distinguish between data-conflict aborts and aborts caused by read-set evictions from L3 cache, and thus both phenomena are reported as conflict aborts. When a thread incurs the maximum number of failed transactions and acquires the lock in the fallback path, it forces all currently running transactions to abort. We term this a *lock* abort. These aborts are reported either as conflict aborts, *explicit* aborts (i.e., deliberately triggered by the application code), or both (i.e., the machine increments multiple counters). Lock aborts are specific to the fallback mechanism we use in our experiments. Other fallback mechanisms that do not use serialization locks within transactions (they can be employed in non-transactional code) do not incur such aborts. Note that avoiding lock aborts does not necessarily result in better performance since the lock aborts mask other type of aborts (i.e., that would have occurred subsequently). This can be seen in abort contributions shown in the figure. As applications are scaled, the fraction of aborts caused by locks increases because every acquisition potentially triggers N-1 lock aborts (where N is the number of threads).

The `RTM_RETIRED:ABORTED_MISC3` performance counter reports aborts due to events like issuing unsupported instructions, page faults, and page table modifications. The `RTM_RETIRED:ABORTED_MISC5` counter includes miscellaneous aborts not categorized elsewhere, such as aborts caused by interrupts. Table III gives an overview of these abort types. In addition to these counters, three more performance counters represent categorized abort numbers: `RTM_RETIRED:ABORTED_MISC1` counts aborts due to memory events like data conflicts and capacity overflows; `RTM_RETIRED:ABORTED_MISC2` counts aborts due to uncommon conditions; and `RTM_RETIRED:ABORTED_MISC4` counts aborts due to incompatible memory types (e.g., due to cache bypassing or I/O accesses). In our experiments, `RTM_RETIRED:ABORTED_MISC4` counts are always less than 20, which we attribute to hardware error (as per the Intel specification update [15]). In all our experiments, `RTM_RETIRED:ABORTED_MISC2` is zero.

V. IMPACT OF PROGRAMMING STYLE ON RTM PERFORMANCE

Transactions are intended to simplify synchronization in multi-threaded programs. Unfortunately, programmers who overlook the best-effort nature of hardware support like RTM may find that their applications do not perform as expected. Programmers may inadvertently make design decisions that limit the ability of the hardware to successfully commit transactions. In this section, we examine two popular transactional applications whose performance can be significantly improved with minimal programming effort — and without finer-grain synchronization — by simply keeping in mind the general constraints of the hardware. We demonstrate that software modifications can help minimize capacity aborts, conflict aborts, and other RTM-unfriendly events.

A. Case Study I: `intruder`

This benchmark emulates a network intrusion detection system that processes packets in parallel. When the first packet in a flow (or session) is captured, a node is inserted into a red-black tree. Subsequent packets are inserted into a list at the node corresponding to the flow. When all packets of a flow are collected, the flow is removed from the tree and pushed into a decoded queue. Flows are subsequently extracted and compared against a database of attack signatures. The main transaction encloses the reassembly phase that inserts packets into the tree of incomplete flows.

Reducing read-set size and transaction duration. The reassembly phase maintains a sorted list of fragments in a flow. Each new packet captured is inserted according to its sequence number. Each reassembly transaction thus traverses a potentially long list to find the correct location. Since program correctness does not rely on keeping captured fragments sorted at all times, we can simply prepend the fragments onto the list in constant time and only sort the list when the flow is complete. We thus reduce both the transaction footprint and the duration for the common case of inserting a packet into an existing flow. The benefits

are twofold: first, shorter-running reassembly transactions reduce the likelihood of conflicts with concurrent tree operations on other flows; and second, accessing fewer cache lines minimizes the likelihood of suffering capacity-induced aborts, allowing more transactions to commit.

Table IV shows results for experiments running the out-of-the-box baseline and our modified version of `intruder` with the recommended large input set. Our simple optimization reduces execution time by almost 50% in all thread configurations, and it reduces the abort rate from 28% to 14% in runs with four threads. Transaction duration is halved, going from about 1800 to about 900 cycles, on average. For the single-threaded configuration, memory-induced aborts (capacity+conflict) for the main transaction (TID1) decline from 86% to 36%. Note that the hardware implementation of RTM may interpret capacity aborts as conflict aborts when passing the status to the abort handler.

B. Case Study II: `vacation`

`vacation` emulates a travel reservation system implemented as an online transaction processing system similar in design to SPECjbb2000 [16]. The database consists of four tables implemented as red-black trees. The customer tree associates customers with their list of reserved travel items, and the three remaining trees contain the reservable items, the associated prices, and the available quantities. Client threads interact with the database in three types of sessions: reservations, cancellations, and updates. Each session is enclosed in a coarse-grain transaction to maintain database validity. Reservation sessions query the item tables to search for the price and availability of a given item and then add reservations to the customer's list (decreasing the number of available instances appropriately).

In our experiments, we scale the database down to 64K relations to reduce capacity aborts, and we run only user sessions (-u 100) to reduce conflict-induced aborts as much as possible. We execute a total of 32M transactions. This workload allows us to better observe the effects of our optimizations compared to a more RTM-friendly baseline.

Reducing read-set size and transaction duration. We find that the programming style used in `vacation` creates unnecessarily long transactions due to redundant tree lookups. For example, in a reservation session, the benchmark searches the tree to check for an item's existence and then again looks up the item to find its price. Similarly, when item reservations are added to a customer's list, items queried in the previous step are looked up again to update their availability. With little extra effort, programmers could merge the queries for availability and price, both of which return references to the found item. The reservation step can use this pointer to directly access items to be booked, obtaining the price and updating availability while avoiding redundant tree searches.

The way elements are placed in data structures can lengthen transactions and increase the sizes of their read-write sets. In `vacation`, the customer reservation list is kept sorted by type of item and ID, and every new reservation traverses the list. The list is never searched for a specific reservation, though, and cancellation sessions do not require any specific ordering of the elements (since they simply iterate through the list to return each booking to the system). We simply change the code to always make insertions at the head of the list, avoiding traversals in the common case (reservations).

Reducing aborts due to page faults. After the transaction has performed all queries in a reservation session, it allocates new memory to insert new reservations into the customer list. Accesses to this newly allocated memory can cause page faults that cause expensive `RTM_RETIRED:ABORTED_MISC3` aborts. If aware of this problem, programmers can improve RTM performance by triggering those page faults before the transaction. It suffices to modify the the STAMP thread-local memory allocator to touch new memory locations before returning.

Table V shows the results of comparing the baseline against our optimized version of `vacation` (which includes the changes described above, applied cumulatively). With rather straightforward changes in the code we reduce execution time by approximately 25% for all thread configurations. Abort rates drop from 21% to 7% in four-thread runs because we eliminate virtually all aborts from page faults (which generally fall under the *HLE-unfriendly instruction* category or `RTM_RETIRED:ABORTED_MISC3`). Transactions are also around 10% shorter. Note that after applying the optimizations, aborts of type `RTM_RETIRED:ABORTED_MISC5` (e.g., from interrupts) become more important as we increase the number of threads.

VI. Related Work

Hardware transactional memory systems must track memory updates within transactions and detect conflicts (read-write, write-read, or write-write conflicts across concurrent transactions or non-transactional writes to active locations within transactions) at the time of access. The choice of where to buffer speculative memory modifications has microarchitectural ramifications, and commercial implementations naturally strive to minimize modifications to the the cores and on-chip memory hierarchies on which they are based. For instance, Blue Gene/Q [8] tracks updates in the 32MB L2 cache, and the IBM System z [9] series and the canceled Sun Rock [6] track updates in their store queues. Like the Haswell RTM implementation that we study here, the Vega Azul Java compute appliance [7] uses the L1 cache to record speculative writes. The size of transactions that can benefit from such hardware TM support depends on the

Table IV
INTRUDER: KEY STATISTICS FOR BASELINE VERSUS OPTIMIZED CODE

intruder		Overall					TID1			
	Threads	Execution Time	% Reduction	Speedup	Cycles/ Transaction	Abort Rate	Abort Rate	% Capacity	% Conflict	% Other
Base	1	15.5	-	1.00	1847	0.13	0.31	0.23	0.63	0.14
	2	8.5	-	1.82	1830	0.19	0.40	0.17	0.64	0.19
	4	4.9	-	3.16	1699	0.28	0.51	0.11	0.60	0.29
Opt	1	7.9	49	1.00	944	0.05	0.13	0.17	0.19	0.64
	2	4.4	48	1.80	959	0.08	0.17	0.10	0.37	0.53
	4	2.7	45	2.93	894	0.14	0.22	0.05	0.48	0.47

Table V
VACATION: KEY STATISTICS FOR BASELINE VERSUS OPTIMIZED CODE

vacation		Overall					Abort distribution		
	Threads	Execution Time	% Reduction	Speedup	Cycles/ Transaction	Abort Rate	% Memory	% HLE-unfriendly	% Other
Base	1	38.8	-	1.00	3360	0.11	0.21	0.76	0.03
	2	20.1	-	1.93	3284	0.14	0.31	0.64	0.05
	4	10.6	-	3.66	3095	0.21	0.43	0.51	0.06
Opt	1	29.4	24.23	1.00	2725	0.02	0.89	0.01	0.10
	2	14.9	25.87	1.97	2720	0.03	0.66	0.02	0.32
	4	7.9	25.47	3.72	2711	0.07	0.13	0.01	0.86

capacity of the chosen buffering scheme. Like us, others have found that rewriting software to be more transaction-friendly improves hardware TM effectiveness [7].

Previous studies investigate the characteristics of hardware transactional memory systems. Wang et al. [8] use the STAMP benchmarks to evaluate hardware transactional memory support on Blue Gene/Q, finding that the largest source of TM overhead is loss of cache locality from bypassing or flushing the L1 cache.

Yoo et al. [10] use the STAMP benchmarks to compare Haswell RTM with the TL2 software transactional memory system [17], finding significant performance differences between TL2 and RTM. We perform a similar study and find that TinySTM consistently outperforms TL2, and thus we choose the former as our STM point of comparison. Our RTM scaling results for STAMP benchmark concur with their results.

Wang et al. [18] evaluate RTM performance for concurrent skip list scalability, comparing against competing synchronization mechanisms like fine-grained locking and lock-free linked-lists. They use the Intel RTM emulator to model up to 40 cores, corroborating results for one to eight cores with Haswell hardware experiments. Like us, they highlight RTM performance limitations due to capacity and conflict misses and propose programmer actions that can improve RTM performance.

Others have also studied power/performance trade-offs for TM systems. For instance, Gaona et al. [19] perform a simulation-based energy characterization study of two HTM systems: the LogTM-SE *Eager-Eager* system [2] and the Scalable TCC *Lazy-Lazy* system [20]. Ferri et al. [21] estimate the performance and energy implications of using TM in an embedded multiprocessor system-on-chips (MPSoCs), providing detailed energy distribution figures from their energy models.

In contrast to the work presented here, none of these studies analyzes energy expenditure for a commercial hardware implementation.

VII. CONCLUSIONS

The Restricted Transactional Memory support available in the Intel Haswell microarchitecture makes programming with transactions more accessible to parallel computing researchers and practitioners. In this study, we compare RTM and TinySTM, a software transactional memory implementation, in terms of performance and energy. We highlight RTM's hardware limitations and quantify their effects on application behavior, finding that performance degrades for workloads with large working sets and long transactions. Enabling hyper-threading worsens RTM performance due to resource sharing at the L1 level. Given these limitations, we show examples of how programmers can optimize TM applications to better utilize the Haswell support for RTM.

ACKNOWLEDGMENTS

This work has been supported by the Swedish Foundation for Strategic Research under grant RIT10-0033.

REFERENCES

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th International Symposium on Computer Architecture*, 1993, pp. 289–300.

[2] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling hardware transactional memory from caches," in *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, 2007, pp. 261–272.

[3] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proceedings of the 31st International Symposium on Computer Architecture*, 2004, pp. 102–113.

[4] M. Lupon, G. Magklis, and A. González, "A dynamically adaptable hardware transactional memory," in *Proceedings of the 43rd International Symposium on Microarchitecture*, 2010, pp. 27–38.

[5] R. Titos-Gil, A. Negi, M. E. Acacio, J. M. Garcia, and P. Stenstrom, "Zebra : A data-centric, hybrid-policy hardware transactional memory design," in *Proceedings of the 25th International Conference of Supercomputing*, 2011, pp. 53–62.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," *ACM SIGPLAN Notices*, vol. 44, no. 3, pp. 157–168, Mar. 2009.

[7] C. Click, "Azul's experiences with hardware transactional memory," 2009, http://sss.cs.purdue.edu/projects/tm/tmw2010/talks/Click-2010_TMW.pdf.

[8] 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 *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 127–136.

[9] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for IBM System z," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 25–36.

[10] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of Intel transactional synchronization extensions for high-performance computing," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 19:1–19:11.

[11] *Intel Architecture Software Developer's Manual: System Programming Guide*, Intel, Jun. 2013.

[12] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2008, pp. 35–46.

[13] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "Eigenbench: A simple exploration tool for orthogonal TM characteristics," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2010, pp. 1–11.

[14] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.

[15] *Desktop 4th Generation Intel Core™ Processor Family Specification Update*, Intel, August 2013.

[16] S. P. E. Corporation, "SPECjbb2000 benchmark," 2000. [Online]. Available: http://www.spec.org/osg/jbb2000/

[17] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *Proceedings of the 19th International Symposium on Distributed Computing*, 2006.

[18] Z. Wang, H. Qian, H. Chen, and J. Li, "Opportunities and pitfalls of multi-core scaling using hardware transaction memory," in *Proceedings of the 4th Asia-Pacific Workshop on Systems*, 2013, pp. 3:1–3:7.

[19] E. Gaona-Ramirez, R. Titos-Gil, J. Fernandez, and M. Acacio, "Characterizing energy consumption in hardware transactional memory systems," in *22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2010, pp. 9–16.

[20] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A scalable, non-blocking approach to transactional memory," in *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, 2007, pp. 97–108.

[21] C. Ferri, R. Bahar, A. Marongiu, L. Benini, M. Herlihy, B. Lipton, and T. Moreshet, "SoC-TM: Integrated HW/SW support for transactional memory programming on embedded MPSoCs," in *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis*, 2011, pp. 39–48.