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 (HTM) implementations are becoming increasingly available. For instance, the Intel Core$^{\text{TM}}$ i7 4770 implements Restricted Transactional Memory (RTM) support for Intel Transactional Synchronization Extensions (TSX). We compare performance and energy expenditures of RTM to those of the TinySTM software transactional memory system by orthogonally varying workload characteristics using Eigenbench microbenchmarks. We show that crossover points exist over the range of workload characteristics for TinySTM and RTM performance and energy figures. We then apply the insights we gather to explain the performance and energy comparison results between RTM and TinySTM for the STAMP benchmark suite. We compare RTM performance overheads to those of other synchronization primitives like spinlock and compare-and-swap (CAS). Finally, we conduct a case study of a few STAMP applications to assess the impact of programming style on RTM performance and 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.

Academic research 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], [3], [4], [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 microarchitecture from Intel to provide such hardware support. Transactional Synchronization Extensions (TSX), the Intel instruction set extensions for HTM, allow programmers to run transactions on a best-effort HTM implementation — the platform provides no guarantees that hardware transactions will commit successfully, and thus the programmer is required to provide a non-transactional path as a fallback mechanism. Intel TSX provides 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's performance and energy 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 excessive energy consumption. We compare RTM performance to TinySTM, a software transactional memory (STM) implementation that uses time to reason about the consistency of transactional data and about the order of transaction commits. We find that although RTM often demonstrates performance advantages, TinySTM outperforms RTM under some conditions. 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 HTM, but as contention increases, RTM consistently performs better. 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

We use an Intel 4th Generation Core$^{\text{TM}}$ i7 4770 processor for our experiments. The four physical cores can run up to eight simultaneous threads with hyper-threading enabled. The processor has 32 KB private L1 cache, 256 KB private L2 cache, and 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 chip energy via the Running Average Power Limit (RAPL) interface. We verify these energy figures against measurements at the ATX CPU power supply input of the motherboard and find them to be quite accurate.[1] 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 carry out microbenchmark studies to assess RTM's limitations. These tests and their results are explained 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 carry out an experiment to 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

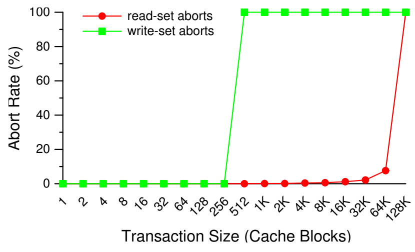


Figure 1. RTM Read-Set and Write-Set Capacity Test

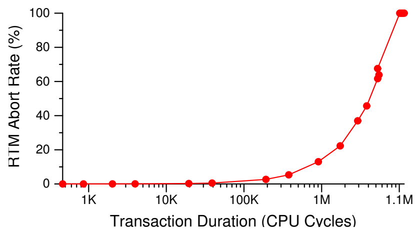


Figure 2. RTM Abort Rate vs. Transaction Duration

of reads within the transaction. Fig. 2 shows the results. Transaction duration begins to affect the abort rate beyond 30K CPU cycles, and when duration reaches more than 10M cycles, all transactions abort (although these figures are likely machine dependent).

RTM Overhead Test. Next we attempt to quantify the performance overheads that RTM incurs compared to spin locks and atomic instructions such as compare-and-swap (CAS). For this comparison, we create a microbenchmark that removes elements from a queue. We first compare RTM against the spinlock implementation found in the Linux kernel (`arch/x86/include/asm/spinlock.h`). For RTM, we simply retry the transaction on aborts. The queue comes from the STAMP [10] library; we modify a version to use CAS in `queue_pop()`. We initialize the queue with 1M elements and let threads extract elements until the queue is empty, with no static division of the work among threads.

We perform three sets of experiments. First, to observe the cost of starting a transaction in RTM when there is no contention, we run experiments with a single thread. We then repeat the experiment with four threads to generate a high-contention workload. Finally, we lower the contention by making threads work on local data for a fixed number of operations after each critical section completes. 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 overheads of RTM versus locks and CAS

[1]We graphed both data sets and verified that the resulting curves have the same shape (although their y-axis scales necessarily differ).

Characteristic	Definition
Concurrency	Number of concurrently running threads
Working-set size	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

Table II
TM CHARACTERISTICS STUDIED

Table I shows that the cost of starting a transaction in RTM is considerable, making it perform worse than the other alternatives when executing non-contended critical sections with few instructions. Compared to using locks and CAS, RTM suffers a slowdown of around 45%; compared to unsynchronized code the slowdown grows to a factor of two. 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. In contrast to locks and CAS, transactions avoid hold-and-wait behavior, which seems to give RTM an advantage for the scenarios analyzed. When comparing locks and CAS in this case, the higher overheads for locks is likely due in part to the ping-pong coherence behavior of the cache line containing the lock and to the cache-to-cache transfers of the line holding the queue head.

B. Eigenbench characterization

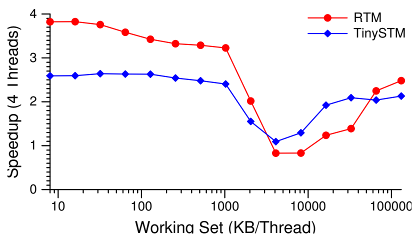
To compare RTM and STM in detail, we next study the behaviors of Hong et al.'s Eigenbench [11]. This parameterizable microbenchmark attempts to characterize the design space of TM systems by orthogonally exploring different transactional application behaviors. Table II defines the seven characteristics we use to compare performance and energy expenditure of the Haswell RTM implementation and the TinySTM [12] software transactional memory system. Hong et al. [11] 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 working set sizes less than 8 MB have little influence on TinySTM's abort rate, 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 efficiencies given are relative to the sequential run of the same size working set.

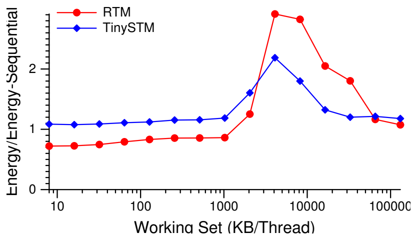
Working-Set Size. Fig. 3 shows Eigenbench results as we increase each thread's working set from 8K to 128MB over a logarithmic scale. RTM outperforms TinySTM for smaller working sets. The performance of both RTM and TinySTM drops once the combined working sets of all threads exceed the 8MB L3 cache. RTM performance suffers more because events like eviction of read-set from L3, page faults, and interrupts trigger a transaction abort, which is not the case for TinySTM. Fig. 3(c) shows a steep rise in RTM abort rates as working sets approache the L3 cache size. The speedups of both RTM and TinySTM are lowest at working sets of 4MB: at this point, the parallelized code's working sets (16 MB in total) exceed L3, but the working set of the sequential version (4 MB) still fits. For working sets above 4MB, the sequential version starts encountering L3 misses, and thus the relative performance of both transactional memory implementations begins to improve. TinySTM's false conflict detection increases sharply when the working-set size reaches 16 MB. For working-set sizes above 64MB, RTM abort rates decrease compared to those for snakker working sets (4-32MB), which makes RTM outperform TinySTM. As for energy efficiency, RTM runs are more energy efficient than both TinySTM and the sequential runs for working sets of up to 1 MB.

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 256 KB working sets, RTM performance drops sharply when the transaction length exceeds 100 accesses. Evicting write-set data from L1 triggers a transaction abort, but when the working set fits within the L1 cache, 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 have a higher likelihood of being evicted. In contrast, TinySTM shows no performance dependence on working set size. The overhead associated with 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.

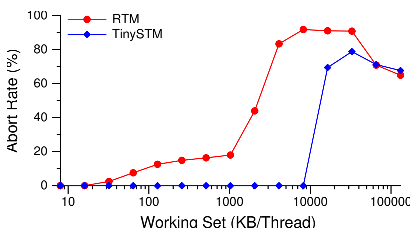
Pollution. Fig. 5 shows Eigenbench results when we test symmetry with respect to handling read-sets and write-sets by gradually increasing the fraction of writes. The pollution level is zero when all memory operations in the transaction are reads and one when all memory operations are writes. When the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



(a) Speedup

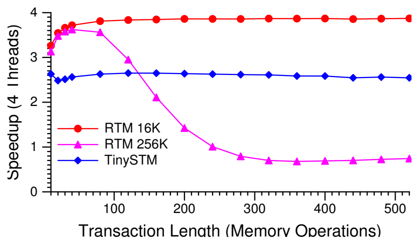


(b) Relative Energy

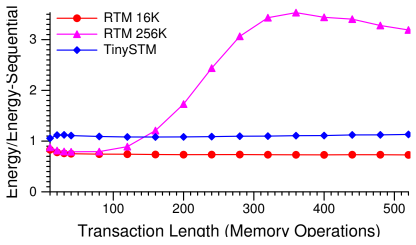


(c) Abort Rate

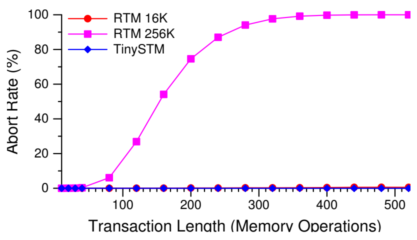
Figure 3. Eigenbench Working-Set Size Analysis



(a) Speedup



(b) Relative Energy



(c) Abort Rate

Figure 4. Eigenbench Transaction Length Analysis

increases. TinySTM outperforms RTM when the pollution level increases beyond 0.4.

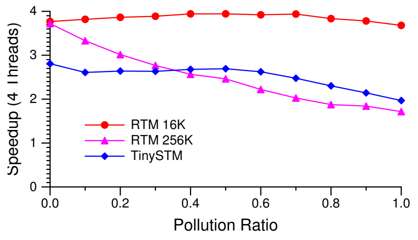
Temporal Locality. We next study the effects of temporal locality on TM performance (where temporal locality is defined as the probability of repeatedly accessing the same memory address within a transaction). The results in Fig. 6 reveal that RTM shows no dependence on temporal locality for the 16KB working set, but performance degrades for the 256 KB 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. For this analysis, we set the working-set size to 64KB per thread. 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. [11]). The conflict probability figures shown here are calculated at word granularity and hence are valid only for TinySTM. Since RTM detects conflict at the granularity of cache line (64 bytes), the contention level is higher for RTM for the same workload configuration. When the degree of contention among competing threads is low, TinySTM considerably outperforms RTM. As contention increases, however, TinySTM performance degrades while RTM performance remains almost the same.

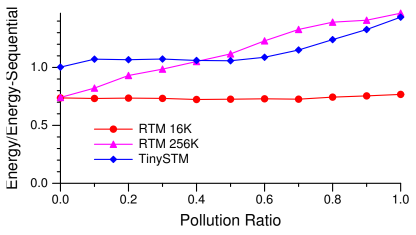
Predominance. We study the behavior of the TM systems when varying the fraction of application cycles executed within transactions to the total number of application cycles. For this analysis, we use the larger working set (256 KB/thread), we set contention to zero, and we vary the predominance ratio from 0.125 to 0.875. Fig. 8 shows that performance for both TinySTM and RTM suffers as the predominance of transactional cycles over non-transactional cycles grows. This can be attributed to the overhead associated with 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 compared to those of the sequential code when concurrency is increased from one thread to eight. Fig. 9 shows that RTM scales well up to four threads. When running 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 hyperthreading 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.

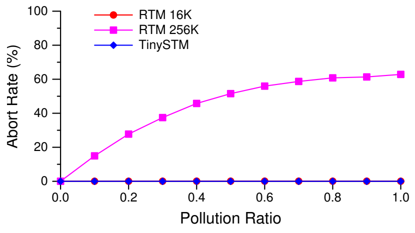
The results from the Eigenbench analysis helps us in identifying a range of workload characteristics for which either RTM or TinySTM is better performing or more energy efficient. We next apply the insights gained from our microbenchmark studies to analyze the performance and energy numbers we see for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



(a) Speedup

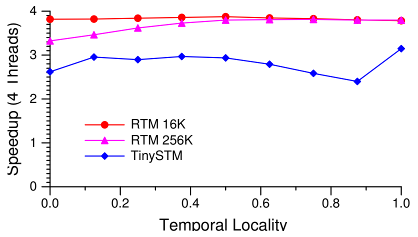


(b) Relative Energy

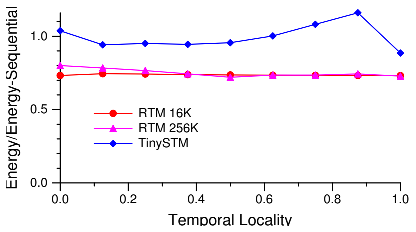


(c) Abort Rate

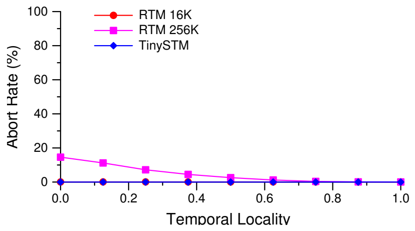
Figure 5. Eigenbench Pollution Analysis



(a) Speedup

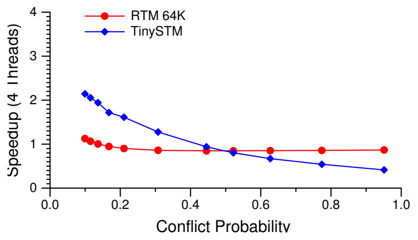


(b) Relative Energy

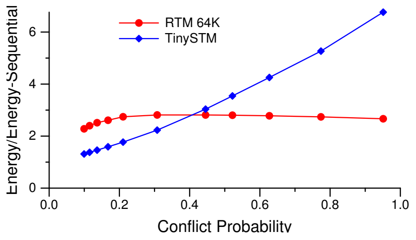


(c) Abort Rate

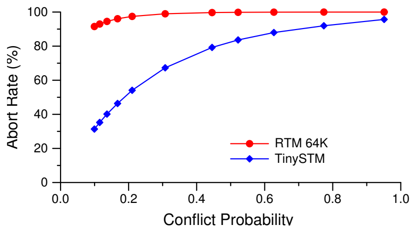
Figure 6. Eigenbench Temporal Locality Analysis



(a) Speedup

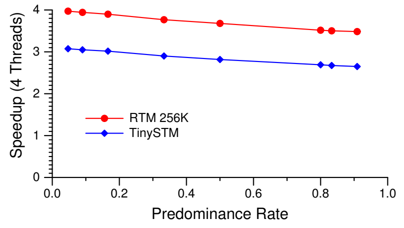


(b) Relative Energy

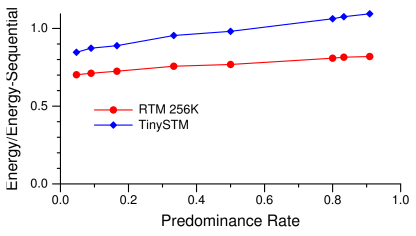


(c) Abort Rate

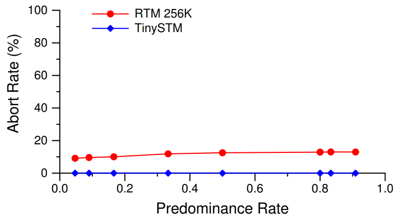
Figure 7. Eigenbench Contention Analysis



(a) Speedup

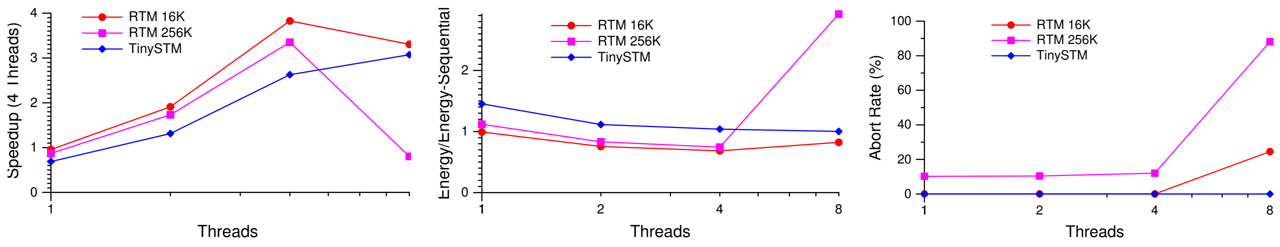


(b) Relative Energy



(c) Abort Rate

Figure 8. Eigenbench Predominance Analysis



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

Figure 9. Eigenbench Concurrency Analysis

IV. HTM versus STM using STAMP

We next use the STAMP transactional memory benchmark suite [10] to compare the performance and energy efficiency of RTM and TinySTM. For this set of experiments, we use the lock-based fallback mechanism explained in section II. We run the benchmarks with input sizes recommended for running on hardware platform that create large working sets and high contention. All results are averaged over 10 runs. Fig. 10 shows STAMP execution times for RTM and TinySTM normalized to the execution time of a sequential (non-TM) run. Fig. 11 shows the corresponding energy expenditures normalized to the energy expenditure of a sequential run. For single-threaded runs, normalized execution time and energy of single-threaded TM versions of the benchmarks illustrate the overheads incurred by the TM systems.

`bayes` has large working set and long transaction length. As we saw from the Eigenbench transaction length analysis in Fig. 4, RTM performs worse compared to TinySTM for applications exhibiting such characteristics. As expected, RTM does not improve the performance of `bayes` as the number of threads scale, and TinySTM performs better overall. Since the execution time taken by algorithm to learn the network dependencies depends on the order of computations, we see significant deviations in learning time for multi-threaded runs.

`genome` exhibits medium transaction length and medium working-set size with low contention. The dominating transaction length in `genome` is less 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 performs better than RTM when contention is low (Fig. 7(a)). The confluence of these two factors within `genome` results in similar performances for both 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 resource limitation issues as it did for `genome`, and thus RTM and TinySTM see similar performance. Even though this application uses a small to medium-sized working set — which might otherwise give RTM an advantage — its performance is dominated by very short transaction lengths that limit speedup.

`kmeans` is a clustering algorithm that groups data items in N-dimensional space into K clusters. Like `bayes`, we see significant deviations in execution times for the multi-threaded versions. On average, RTM performs better than TinySTM. The short transactions experience low contention, and the `kmeans` working set size is small with high locality, all of which work together to give RTM a performance advantage over TinySTM. Even though both TM systems show speedups over the sequential run, only RTM saves energy: 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. The results in Fig. 10 show that labyrinth does not scale in RTM. This is because each thread makes a copy of global grid inside the transaction, triggering capacity aborts that eventually cause the transaction to fall back 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.

`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 a big margin, as expected for very short transactions.

`vacation` has low to medium contention among threads and 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 the performance degrades for eight threads since its read-write set size is not small enough to avoid resource limitation

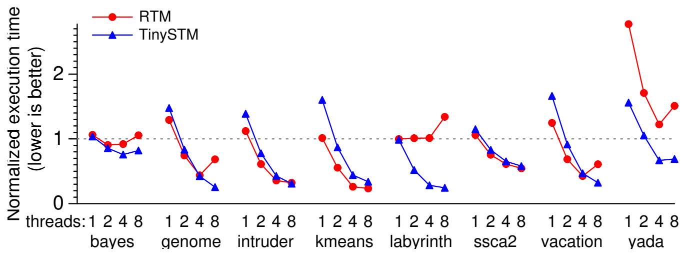


Figure 10. RTM vs TinySTM performance for STAMP benchmarks

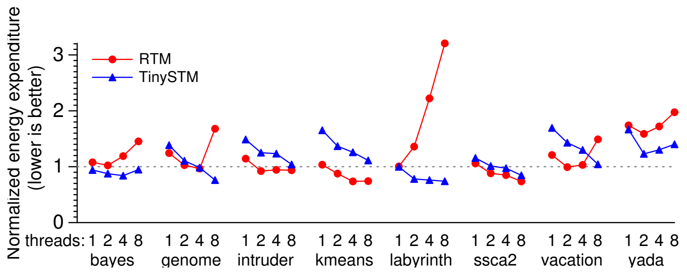


Figure 11. RTM vs TinySTM Energy Expenditure for STAMP benchmarks

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

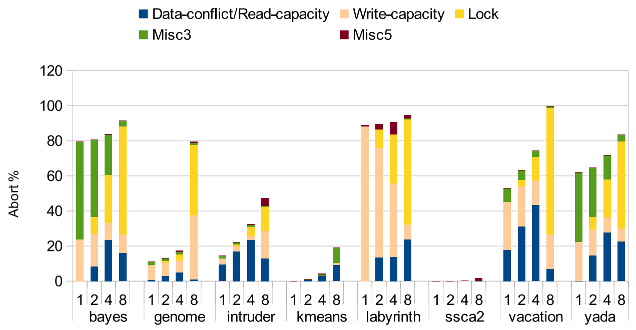


Figure 12. RTM Abort Distribution for STAMP benchmarks

issues arising from cache sharing.

`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.

A common observation from the STAMP results is that for the applications that have large read-write set size and large working-set size (`bayes`, `labyrinth` and `yada`), the energy trends do not follow performance trends. These applications expend more energy as they are scaled to more threads — possibly due to increased cache and bus activity — even when the performance remains constant or falls.

Fig. 12 shows the overall abort rates for all benchmarks, including the contributions of different abort types. As per our observations of hardware counter values, the current 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 the resulting abort 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). Such 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 suffer such aborts. Note that avoiding lock aborts does not necessarily result in better performance since the lock aborts mask other type of aborts. This can be seen in abort contributions shown in Fig. 12. As the application is scaled, the fraction of aborts caused by lock increases. This is because every lock acquisition by a thread would potentially cause $N - 1$ lock aborts where N is the number of threads. If however, the lock was read outside transaction, the transactions that were aborted due to lock acquisition, would be allowed to continue and may abort due to data conflict or capacity overflow later.

The `RTM_RETIRED:ABORTED_MISC3` performance counter reports aborts due to unsupported instructions, page faults, page table entry modifications, etc. The `RTM_RETIRED:ABORTED_MISC5` counter includes miscellaneous aborts not categorized elsewhere, for example, 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 (values for which we always observed to be zero value in our experiments); and `RTM_RETIRED:ABORTED_MISC4` counts aborts due to incompatible memory types (e.g., due to cache bypass or I/O access). Our observed values for the latter counter were always less than 20; we attribute this to hardware error, as per the Intel specification update [13].

V. Impact of Programming Style on RTM performance

Transactions are intended to simplify synchronization in multithreaded programs. Unfortunately, the truth is that programmers who overlook the best-effort nature of the hardware where their applications run, may often find that their applications do not perform as expected. Programmers may inadvertently make design decisions that greatly 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 — without finer-grain synchronization — simply by keeping in mind the general constraints of the hardware. We demonstrate that it is possible to minimize the occurrence of 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. Packets are first captured from a stream, then reassembled into a complete flow and finally compared against a known set of intrusion signatures. In the reassembly phase, a red-black tree is used to keep captured packets: each node of the tree contains a list of packets belonging to the same flow (session). Flows are inserted at the point the first packet is captured. For subsequent packets, `intruder` finds the correct flow in the tree and inserts them in the corresponding list. When all packets of a flow have been collected, the flow is reassembled, 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 in which a captured packet is inserted 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, so each new packet captured is inserted into position according to its sequence number. Each reassembly transaction thus traverses a potentially long list to find the correct insertion 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 prior to reassembly. We thus reduce both the transaction footprint and the duration for the common case of inserting into an existing flow. The benefits are twofold: on the one hand, shorter-running reassembly transactions reduce the likelihood of conflicts with concurrent tree operations on other flows; and on the other hand, accessing fewer cache lines minimizes the likelihood of suffering capacity-induced aborts, which allows more transactions to commit in a best-effort HTM implementation.

Table IV shows results for experiments running the baseline and our optimized versions of `intruder` with the recommended large input set. Our simple optimization

Table IV
INTRUDER: KEY STATISTICS FOR BASELINE VERSUS OPTIMIZED CODE

intruder		Overall					TID1			
	Threads	Exec. Time	% Reduc	Speedup	Cycles/ Tx	Abort Rate	Abort Rate	% Capac	% Confl	% 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

reduces execution time (*Exec. Time*, shown in seconds) by almost 50% in all thread configurations (*% Reduc* column), and it reduces the abort rate from 28 to 14% in runs with four threads. Transaction duration is halved, going from around 1800 to 900 cycles, on average. For the single-threaded configuration, we observe that 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 being of conflict type when passing the abort 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 [14]. 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 travel items, the associated price, and the available quantity. 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 the validity of the database. 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 size down to 64K relations to reduce capacity aborts to reasonable levels, 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 baseline that is as RTM-friendly as possible.

Reducing read-set size and transaction duration. We find that the programming style used in vacation creates transactions of unnecessarily long duration 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 reservation 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. Extra customer table lookups will thus be avoided.

The way elements are placed in data structures can have lengthen transactions and increase the size of their read-write set footprints. In vacation, the customer reservation list is kept sorted by type of item and id, and every new reservation needs 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 (reservation sessions).

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 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
VACATION: KEY STATISTICS FOR BASELINE VERSUS OPTIMIZED CODE.

vacation		Overall					Abort distribution		
	Threads	Exec. Time	% Reduc	Speedup	Cycles/ Tx	Abort Rate	% Mem	% HLE-unfr	% 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

Table V shows the results of our comparison between the baseline benchmark against an optimized version of vacation, which includes the changes described above, applied accumulatively. As can be seen in the table, with rather straightforward changes in the code a programmer can reduce execution time by approximately 25% for all thread configurations. Abort rates drop from 21 to 7% in 4-threaded runs, thanks to the elimination of virtually all aborts caused by page faults (which generally fall under the *HLE-unfriendly instruction* category or misc3). Transactions are also around 10% shorter. Note that after applying the optimizations, aborts of type misc5 (e.g., 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 cancelled 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 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 have investigated 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. [15] use the STAMP benchmarks to compare Haswell RTM with the TL2 software transactional memory system [16], finding significant performance differences between TL2 and RTM. We performed a similar study and found 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 those of Wang et al. [8].

Others have also studied power/performance tradeoffs. For instance, Gaona et al. [17] 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 [18]. Ferri et al.[19] 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 emergence of Hardware Transactional Memory (HTM) support in commonly available microprocessors is making it attractive for large numbers of programmers to use transactional memory. However, the best-effort nature of these early TM systems can create performance issues. HTM transactions are expected to be used with runtime systems that guarantee forward progress when transactions fail to commit. Carefully avoiding unnecessary serialization in such systems is essential for performance and energy efficiency. In this study, we measure the performance and energy costs of several schemes for shared memory synchronization. We show that HTM does not perform well across the board: Software Transactional Memory (STM) implementations like tinySTM deliver highly competitive performance for certain workloads. We also characterize RTM performance along several dimensions. This should help programmers know what to expect when designing or porting applications to run on RTM. Our case studies highlight common optimizations that programmers should be mindful of to avoid unexpected performance losses.

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," *SIGPLAN Not.*, 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*, ser. PACT '12. New York, NY, USA: ACM, 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*, ser. MICRO '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 25–36.

[10] 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.

[11] 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*, ser. IISWC '10. IEEE Computer Society, 2010, pp. 1–11.

[12] 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.

[13] *Desktop 4th Generation Intel Core^{TM}Processor Family Specification Update*, Intel, August 2013.

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

[15] R. Yoo, C. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of Intel transactional synchronization extensions for high performance computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2013.

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

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

[18] 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.

[19] 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 *2011 Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis*, 2011, pp. 39–48.