Study of Hardware Transactional Memory Characteristics and Serialization Policies on Haswell

Marcio Machado Pereira[a], Matthew Gaudet[b], J. Nelson Amaral[b], Guido Araujo[a]

[a]*Computer Science Institute, University of Campinas — UNICAMP — Brazil*
[b]*Computer Science Departament, University of Alberta, Canada*

Abstract

This paper presents an extensive performance study of the implementation of Hardware Transactional Memory (HTM) in the Haswell generation of Intel x86 core processors. It evaluates the strengths and weaknesses of this new architecture by exploring several dimensions in the space of Transactional Memory (TM) application characteristics using the Eigenbench [1] and the CLOMP-TM [2] benchmarks. This paper also introduces a new tool, called htm-*pBuilder* that tailors fallback policies and allows independent exploration of its parameters.

This detailed performance study provides insights on the constraints imposed by the Intel's Transaction Synchronization Extension (Intel's TSX) and introduces a simple, but efficient policy for guaranteeing forward progress on top of the best-effort Intel's HTM which was critical to achieving performance. The evaluation also shows that there are a number of potential improvements for designers of TM applications and software systems that use Intel's TM and provides recommendations to extract maximum benefit from the current TM support available in Haswell.

Keywords: Programming Techniques, Concurrent Programming, Transactional Memory.

1. Introduction

This paper presents a performance evaluation of the Hardware Transactional Memory (HTM) capabilities in Intel's Haswell micro-architecture called the Transactional Synchronization Extensions (TSX) [3]. The goal is to study performance from the application perspective, providing a precise evaluation of the strengths and weaknesses of this architectural feature. To efficiently exploit the parallelism available through Intel's TSX, it is important to know the constraints imposed on software by its hardware design, and the requirements that must be fulfilled by software support systems.

The main finding of this performance study is that Intel's TSX performance is most sensitive to i) the *transaction footprint*, defined as the number of shared write accesses executed inside a transaction; ii) the *working-set size*, defined as the number of distinct memory locations accessed — read from or written to — inside a transaction;[1] iii) the *transactional write ratio* (or *pollution*), defined as the ratio between the number of shared writes and the total number of shared accesses in a transaction; and iv) the *contention level* in a transactional application. The contention level is the probability that dynamic transactions will abort due to a conflicting access.

This sensitivity is first demonstrated through an analysis that isolates the effect of each TM application characteristic on performance using Eigenbench [1] and CLOMP-TM [2] to identify the constraints imposed by TSX. Next the study evaluates the performance of TSX using the more realistic STAMP benchmark suite [4].

Based on the experimental results, the following application features are likely to yield performance gains when using Intel's TSX:

- In a low-contention scenario, if the *transactional footprint* is small, then it is preferable to convert multiple transactions into a single transactional region. This conversion amortizes the overhead necessary to execute a transaction as long as the *transaction footprint* do not exhaust the capacity of cache lines. In high-contention scenarios, it is preferable to maintain transactions with smaller footprint to reduce wasted work.

- *Transaction footprint* and *contention* are the most important characteristics that dictate TM performance. These characteristics are strongly influenced by *temporal locality* and *pollution*. To obtain performance gains, a designer

Email addresses: mpereira@ic.unicamp.br (Marcio Machado Pereira), mgaudet@ualberta.ca (Matthew Gaudet), amaral@cs.ualberta.ca (J. Nelson Amaral), guido@ic.unicamp.br (Guido Araujo)

[1] A more precise metric is the number of distinct cache lines that are occupied by the working set of a transaction.

should create data structures that increase temporal locality and reduce pollution. For example, data structures based on hash often have a better performance than linked lists because they tend to reduce pollution.

- The performance of Intel's TSX is also sensitive to the policy applied on the fallback path to ensure forward progress. The right choice of fallback policy can avoid unnecessary serialization thereby allowing more concurrency and improving performance.

This paper is organized as follows. Section 2 provides background on Hardware Transactional Memory and introduces the Intel's Transactional Synchronization Extensions with a brief description of transactional abort's causes and the various sources of data conflicts. Section 3 describes the policies generally used on the evaluations, and also introduces *SerControl*, a simple and effective serialization policy that is more efficient for TM on Intel's TSX than policies used so far [5, 6]. Section 4 describes our experimental setup, a detailed assessment of Intel's TSX and recommendations on how to best use the new Intel HTM support. Section 5 presents an evaluation of Intel's TSX using the STAMP benchmark suite [4]. Section 6 explores, with the aid of a new tool called htm-*pBuilder*, the performance of Intel's TSX for various fallback policy tunings and transaction properties. Section 7 discusses related work.

2. Background Information

Transactional Memory (TM) makes it easier for programmers to develop parallel programs. With TM, programmers enclose a group of instructions within a transaction to execute the instructions in an atomic and isolated way. The underlying TM system runs transactions in parallel as long as they do not conflict. Two transactions conflict when they access the same address and one of them writes to it. A Hardware TM (HTM) system uses dedicated hardware to accelerate transactional execution [7, 8, 9]. The HTM system starts a transaction by executing a register checkpoint with shadow register files. Whenever the transaction writes to memory, the transactional value produced by the write is stored separately from the original value by either buffering the transactional data in hardware buffers, such as the cache, or by logging the old value, a process called *data versioning*. Data versioning can be implemented by augmenting the cache with additional bits [10] or by using separate hardware structures, such as Bloom filters [11], to record the memory addresses read and written by the transaction. A conflict between two transactions is detected by comparing the read sets and the write sets of both transactions. If a conflict is detected, one of the transactions is rolled back by discarding its transactional writes, restoring the values in the registers to the values saved at the checkpoint at the start of the transaction, and discarding any transactional changes to the state of the program. When there is no conflict, the transaction commits the transactional data and discards both the transactional metadata and the values saved at the register checkpoint.

Software support for HTM systems is limited by the information and control provided by the hardware-software interface. Typically HTMs offer limited scope to implement contention management, used to provide forward progress in STM systems, and so must rely on a lock based fallback policy to provide forward-progress guarantees. For instance, the TM runtime in the IBM BG/Q machine adopts a policy of performing a certain number of retries for an aborting transactions before causing the transaction to serialize through the acquisition of a global lock [12].

Intel Transactional Synchronizations Extensions (Intel's TSX) [3] is a recent addition to the Intel architecture that provides programmers with hardware transactional memory in the Haswell processor. Intel's TSX provides two software interfaces to programmers: **Hardware Lock Elision** (HLE), a legacy-compatible instruction set extension, and **Restricted Transactional Memory** (RTM), a new instruction-set interface. HLE provides a prefixed instruction that indicates that a lock acquisition is to be elided, with the body executed in a transaction instead. If the transactional execution fails, the execution falls back to the original lock. The RTM interface executes code in a transaction, but provides no guarantee that a transactional execution will eventually commit. Therefore the program must always provide code to handle a transactional abort that can either restart the transaction or take a non-transactional path.

A processor can perform a transactional abort for numerous reasons. A primary cause is conflicting data accesses between transactions executing in different logical processors. Such conflicting accesses force an abort to ensure the preservation of transactional isolation. Transactional aborts may also occur because of limited transactional resources. For example, the amount of data accessed in a transactional region may exceed the HTM capacity limit. Intel's TSX uses the EAX register to communicate abort status to software. Causes for abort in TSX include execution overlap of transactional and non-transactional regions and system events such as system calls and page faults. During startup transactional programs experience a higher rate of aborts due to page faults. They experience a lower rate of such aborts after reaching steady state. However, in programs with very short run times, page-fault-induced aborts may appear to dominate.[2] A high rate of page-fault-induced aborts is also observed soon after large regions of memory are allocated.

3. Forward-Progress Policies

Intel's TSX does not guarantee that a transactional execution will eventually commit. The expectation is that a software

[2]This type of abort affects most of the experiments with current benchmarks.

fallback handler is provided by the policies that guide the TM application. This handler may create an unbounded TM system, similar to a Software TM (STM), by providing forward-progress guarantees for transactions that fail. Generally, the strategy adopted is to retry the execution of the transaction, with or without a time delay, to attempt to complete the transaction execution speculatively. A time delay, often called *backing-off*, can be used to avoid the pathological pattern called *Convoy* [13], where many transactions retry simultaneously. In the face of persisting failure, a transaction must be completed by running it in a non-speculative execution mode. A common solution is to acquire a global lock to prevent other transactions from committing concurrently. The experimental tool described herein implements three forward-progress policies:

MaxRetry is the simplest way to ensure forward-progress. It simply limits the number of times that a dynamic transaction can be retried to a predefined threshold *N*. A transaction that exceeds its retry budget must acquire a global back to execute while. For the benchmarks studied, an empirical evaluation indicated that the best results occur with *N* equal twenty. *MaxRetry* is not a policy used in any actual HTM system. It is included in this study to enable an evaluation of the effect of backing-off in reducing the Convoy pathology described by Bobba *et al.* [13].

Backoff is a forward-progress policy similar to MaxRetry, except that the aborted transaction waits for a time delay before restarting. The delay duration is chosen uniformly at random from a range whose size increases exponentially with every restart. This is continuously done up to the limit of (predefined) *N* consecutive aborts. We conducted experiments for different values of *N* and the best results were obtained for *N* equal to twenty. After that, the transaction must acquire the global lock to execute in exclusive mode.

SerControl is a new forward-progress policy that selects one of three actions upon a transaction abort: *retry*, *backoff* or *serialize*. The *SerControl* policy examines the return code provided by the hardware in the EAX register, which contains various status bits indicating the cause of the abort. If the transaction aborted because of a conflict with other transactions or because of capacity limitations, the action selected is *backoff*. This strategy makes sense because a capacity abort may be caused by competing transactions vying for the same storage resources. To prevent frequent aborts due to capacity overflow *SerControl* serializes a transaction that has suffered two consecutive capacity aborts.[3] If the cause of an abort is other than conflict or capacity, the action is *retry* for three consecutive aborts[4] before changing to *backoff*. The idea is that aborts, such as an abort caused by a page-fault, may not occur again if the transaction is retried immediately. The *SerControl* policy limits retries to a threshold of *N* consecutive aborts. The value of *N* used in the evaluation was twenty for a fair comparison with the other two policies. After *N* retries, a transaction must be executed under the control of a lock — a *serialize* action.

Experience with STM, where transactions are implemented entirely in software, has demonstrated the simplicity of transactional programming, but has raised challenging performance issues [20]. In our experiments with STAMP benchmark suite, we have confirmed the results found in [20]; although Intel's TSX exhibited lower overhead when compared with some STM's (*e.g.*, TinySTM [21]) this is not always true. To achieve better performance, the *SerControl* policy kept this overhead low by using the minimum data structure required to manage a transaction. This data monitors the number of retries and the number of capacity-induced aborts, and selects the *SerControl* policy action and the time delay (see the pseudo-code in Listing 1). Henceforth, the TM overhead measured in the experiments includes the overhead imposed on the transaction management by the forward-progress policies.

Listing 1: Low-Overhead High-Accuracy *SerControl* Policy

```
upon TM_START
  int backoff = MIN_BACKOFF;
  int (try = 0, status = 0);
  int (conflict = 0, capacity = 0);
  long wait = 0;

upon TM_BEGIN
  if (++try>=MAX_RETRY) status = SERIALIZE;
  else
    case (status = _xbegin())
      XBEGIN_STARTED: ;
      XABORT_CONFLICT:
        capacity = 0;
        status = (++conflict>=MAX_CONFLICT) ? BACKOFF : RETRY;
      XABORT_CAPACITY:
        status = (++capacity>=OVERFLOW_CAPACITY) ? SERIALIZE : RETRY;
      others:
        capacity = 0;
        status = (try>=MAX_CONTROL) ? BACKOFF : RETRY;
    esac
  fi
  if (status==SERIALIZE)
    status = get_spin_lock()
  fi
  if (status==IN_SPIN_LOCK) or
     (status==XBEGIN_STARTED and spin_lock==FALSE)
    begin_transaction()
  fi

upon TM_END
  if (status==IN_SPIN_LOCK)
    release_spin_lock()
  else
    commit_transaction()
  fi

upon TM_ABORT
  if (status==BACKOFF)
    wait = get_new_time_delay()
    while (wait--) ;
    backoff++;
  fi
```

Sections 4, 5, and 6 present results from an extensive experimental evaluation based on Eigenbench, CLOMP-TM, and the STAMP benchmarks. In Section 4 transaction properties are varied but the parameters for each serialization policy, such as the maximum number of retries, are constant. Section 5 uses the STAMP benchmark for a similar evaluation with fix policy parameters. Then in Section 6 a new tool is used to explore a range of values for the parameters of the serialization policies.

[3]This safeguard is needed because an abort reported as a capacity abort may be due to temporary competition for resources.

[4]This threshold was determined through a series of experiments, with the STAMP benchmarks, that examined performance, serialization rate, and number of aborts due to different causes.

4. Experimental Evaluation

This section presents an experimental evaluation using a prototype implementation of the *SerControl*, *MaxRetry* and *Backoff* forward-progress policies on top of Intel's TSX processor to guide TM applications. The main findings about the policies are:

- Simple policies, such as *MaxRetry* and *Backoff*, have the potential to deliver performance in RTM because of their low overhead to monitor transaction execution and of their simple approach to decide when to retry an aborted transaction. However, the experimental results indicate that these policies are also sensitive to the tuning of the parameters used to decide when to restart or serialize the execution of a transaction.

- Among the policies evaluated, *SerControl* is the most successful in delivering performance for transactional applications for RTM due to its strategy of reducing potential conflicts and, consequently, the number of aborts. Moreover, strict adherence to Bobba's suggestion of exponential backoff [13] can be detrimental to some applications.

4.1. Experimental Infrastructure and Methodology

This experimental evaluation uses a computer with an Intel Xeon Processor E3-1200 v3 (4 cores with 2 hyper-threads per core, 8 threads in total) clocked at 3.10 GHz with 8 GB of RAM. Each core has a 32 KB L1 Data Cache and a 256 KB L2 Cache. The operating system is Ubuntu Server 13.10 amd64.

The results presented in all experiments are averaged over ten executions and the graphs show a 95% confidence interval. In the graphs that show an upper-bound speedup, this bound represents the case of a multi-threaded execution without the protection of a TM system. This measurement serves as a hypothetical upper limit for the available performance and is useful to show the overhead of the TM system when independent transactions are running (zero contention). In, the experiments that estimate the upper-bound speedup, the atomic execution of critical sections is not guaranteed and the results could be incorrect. Therefore, it is said that the system is executing in unprotected mode. Unless noted, all speedups are normalized to the execution time of the corresponding best sequential version.

4.2. EigenBench Results

Experiments with the Eigenbench [1] micro-benchmark enable a characterization of Intel's TSX RTM and provide significant insight on its strengths and weaknesses. The Eigenbench is designed to enable independent exploration of the properties [5] of a TM application shown on Table 1. Each instance of the Eigenbench benchmark can be thought of as a point in a multi-dimensional space defined by these properties.

Table 1: TM properties

Property	Definition	empirical values
Transaction length	Number of shared accesses per TX	30 words
Working-set size	Size of frequently used memory	32 KB/thread
Pollution	Fraction of shared writes to shared accesses	10%
Temporal locality	Probability of repeated address per shared access	0 %
Contention	Probability of conflict of a transaction	0 %
Predominance	Fraction of shared access cycles inside TX (not explored)	100%
Density	Fraction of non-shared cycles outside TX (not explored)	100%

The methodology to use Eigenbench to discover characteristics of the Intel's TSX RTM implementation consisted of an initial empirical exploration of the space defined by the values of the properties listed in Table 1. This exploration led to the discovery of a "baseline configuration" where the values listed on the rightmost column of Table 1 were used. With these values fixed, a more systematic exploration of the space varied the value of a single property at a time while maintaining the other values at the baseline configuration. In occasions where this exploration indicated that there was potential for better performance when two or more properties were moved away from this baseline, those combinations were also tried.

The graphs in Figure 1 display the trends for each of these experiments using the three policies: *MaxRetry*, *Backoff* and *SerControl*. The horizontal axis denotes the value of the property that is being controlled and the vertical axis denotes the speedup results for four threads, normalized in relation to the sequential execution of the same program. Except for the experiment reported in the graph in Figure 1(g), independent transactions are used to evaluate the overhead of the TM system. To show the overhead, a curve for a performance upper bound is also included. This upper bound is estimated by running the sequential version in parallel "unprotected" with no transactional overhead from fallback-handling code.

An important limitation of HTM implementations is the amount of speculative state that the HTM system can store. RTM can only successfully complete speculative transactions that have a small footprint because all speculative state is stored in the relatively small L1 data cache. The experiment reported in Figure 1(a) demonstrates this limitation. The *SerControl* policy performs better than *MaxRetry* and *Backoff*for a pollution of 1% thanks to its strategy of reducing potential conflicts and hence imposing a lower demand on the L1 data cache in comparison with the other policies. However, the *SerControl* performance degrades quickly for transaction lengths greater than 60 words[6] for a pollution of 10%. The lower performance for smaller transactions (left side of the plots) indicates that the TM overhead becomes significant. A separate measurement, executing on a single thread, confirms this observation. For an

[5]For the Eigenbench, transaction footprint is a derived property, calculated from *transaction length* $\times$ *pollution*.

[6]One (1) word equals four (4) bytes in the target machine.

empty transaction, or a transaction length equal zero, the transactional execution time is twelve times higher than the sequential execution that has no overhead. For transaction length equal to 5 words this time drops to 2 times of the sequential execution. When transaction length reaches 30 words, this overhead is almost totally amortized. The graph on the left in Figure 1(b) shows that *SerControl* perform better than *Backoff* between 60 and 140 words while the graph on the right in same figure points out that *SerControl* is dramatically more reluctant to serialize when capacity aborts occur, confirming one of the payoffs of the *SerControl* policy.

The graphs in Figure 1(c) show the effect of different *working-set* sizes for transaction lengths equals to 30 (left plot) and 50 (right plot) words. *MaxRetry* is the policy that suffers most due to the high number of aborts and retries and hence the overhead is more pronounced in short transactions. There is a dramatic speedup drop starting at 256 KB/thread — even for the upper-bound curve — because the data processed by the transactions exceed the capacity of the L2 cache. For TX length equal thirty the most interesting comparisons are the curves for *MaxRetry* and *Backoff* on the left of the plot and the curves for *Backoff* and *SerControl* on the right of the plot. To better understand these performance trends, Figure 1(d) reports the ratio between the number of transactions that abort and the number of transactions that commit on the left, and the ratio between the number of transactions that serialize and the number of transactions that commit on the right. For smaller working sets, always backing-off is a good policy because it eliminates the convoy-effect aborts resulting from *MaxRetry*, as the abort ratios on the left plot of Figure 1(d) indicate. As the working-set size becomes larger, the selective back off used by *SerControl* is a better policy. However, with longer transactions and larger working-sets, selectively backing-off does not prevent the increasing number of aborts and serializations as shown by the plots on the right of Figures 1(c) and 1(d).

A study of the effect of temporal locality on performance is shown in Figure 1(e). The probability of address repetition varies between 0 (no address is used twice in a transaction) and 1 (a single address is used inside each transaction). The initial drop in performance for both *Backoff* and *SerControl* for a transaction length equal 50 occurs as addresses start being repeated. This may be explained by the associative effect of the L1 data cache — the EigenBench generates random addresses that might cause more potentially useful data elimination from the cache.

Until the *transaction footprint* exhausts cache lines, at about 32% of shared writes for a transaction length of 50 (i.e., 64 bytes), the *SerControl* strategy of avoiding potentially unproductive retries yields higher speedups than either *MaxRetry* or *Backoff* (see Figure 1(f)). The drop in speedup for *SerControl* for larger transactions is due to the limited capacity in the cache to store speculative state.

Although critical details are not available, there is sufficient information to speculate about the nature of Haswell's TM. Haswell's TM uses the L1 data cache to track the write-set. While it stores transactional reads in the L1 data cache, it also uses a separate mechanism, perhaps a *Bloom Filters* [14], to track speculative reads that have been evicted from the L1 data cache. The caches and fill buffers are competitively shared by any active threads. However, store operations only need to write the address (and eventually the data) into the store buffer while load operations must write into the load buffer and also probe the store buffer to check for any forwarding or conflicts. It appears that the shape curve of *MaxRetry* policy on the left graph in Figure 1(f) is due to the difference in the conflict detection mechanism between read- and write-sets and is stimulated by increased competition due to the Convoy effect.

Finally, how does contention affect performance in the Intel's TSX? Experiments with transaction lengths equal to 20, 30 and 40 words — varying the fraction of writes per transaction between 5% and 100% — help address this question. The x-axis shows the value of expected contention. The graphs in Figure 1(g) indicate that the TM performance drops quickly with long or dirty transactions. All three policies are very sensitive to contention greater than 4%, although the *SerControl* policy performs better if the transaction length fit between 30 and 40 words. This level of contention was achieved for write-ratio of 30% to 40%, which implies in a transaction footprint between 48 and 64 bytes. This reveals a severe resource constraint for TSX, limiting the footprint to the size of the L1 data cache.

4.3. CLOMP-TM Results

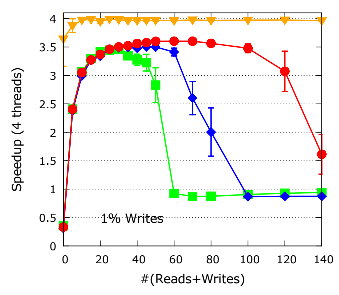
Section 4.2 showed that, to make an effective use of RTM on Intel's TSX, we need to define the design space that can be exploited by the TM applications. For this, it is necessary to complement this study with a more appropriate analysis of the performance behaviour of RTM at different TM footprints. The CLOMP-TM benchmark [2, 5] is used for this analysis.

CLOMP-TM is a synthetic memory-access generator that emulates the synchronization characteristics of high-performance computing applications. It is specifically designed to expose the range of properties needed to characterize scientific workloads. It was created to mimic the application characteristics of several large scale, multi-physics applications used in production at the Department of Energy laboratories in the USA.

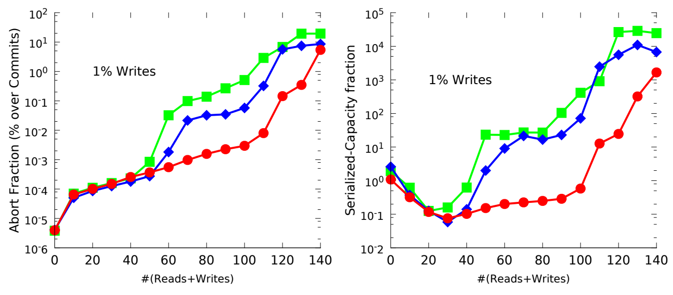
CLOMP-TM uses constructs such as atomics or OpenMP-based constructs (*omp critical* and *omp atomic*) to synchronize OpenMP threads with the same level of abstraction. We adapted CLOMP-TM to use Intel's TSX RTM with the prototype implementation of the *SerControl* policy to evaluate its performance on small and large footprints, and also to compare the performance obtained with the fine-grained lock implementation in both low- and high-contention scenarios.

Figure 1: Analysis of RTM using EigenBench

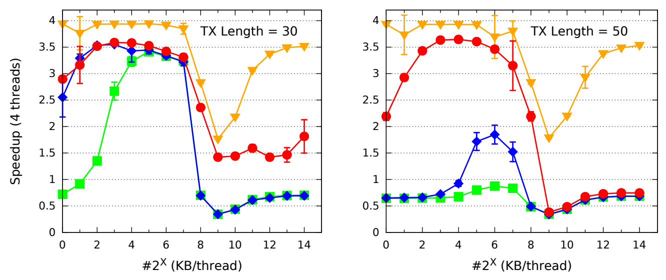
(a) Transaction Length



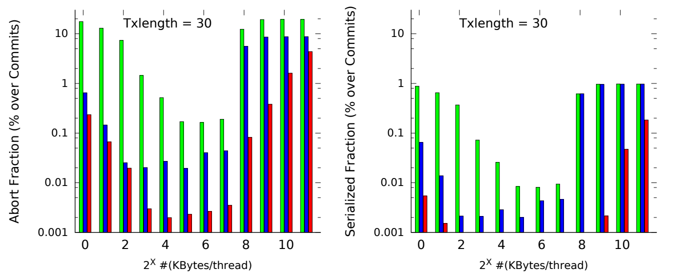
(b) Analysis of outcomes for Transaction Length



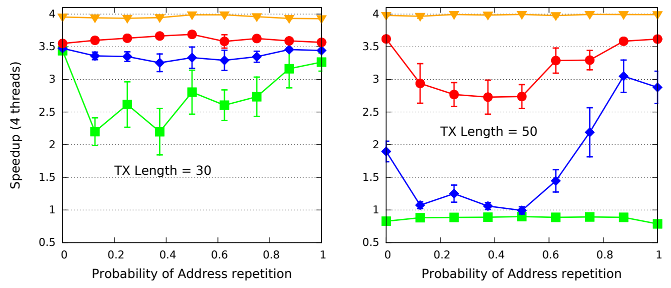
(c) Working-set size



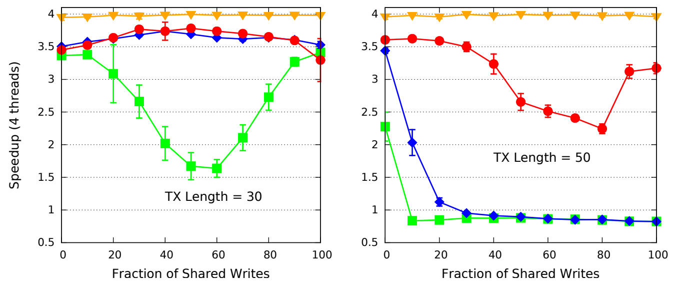
(d) Analysis of outcomes for Working-set



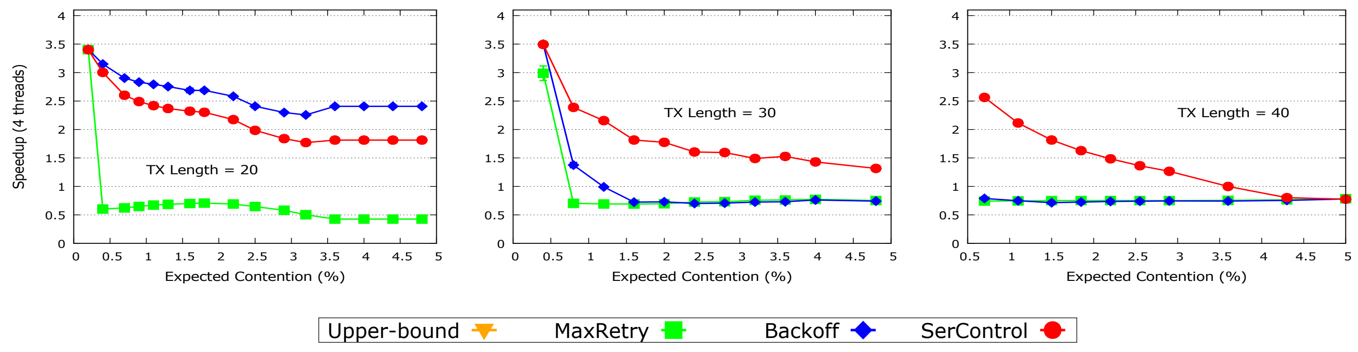
(e) Temporal Locality



(f) Pollution



(g) Contention



CLOMP-TM resembles an unstructured mesh with a set of partitions. Each partition holds a linked list of zones. To vary the pressure on the memory system, the size of these zones were configured with two values: 64 bytes, to fit the size of the cache line, and 128 bytes. Also, two different contention levels were used: low contention and high contention. Contention occurs when multiple threads update the same zones (for the contention scenarios, the memory access patterns used were *Adjacent* and *FirstParts*, respectively.[7]) Each zone is pre-wired to deposit a value to a set of other zones, called *scatter zones*, which involves reading the coordinate of a scatter zone,

[7]For more information on memory access patterns [2].

doing some computation, and depositing the new value back to the scatter zone. These value deposits were synchronized in two ways:

- `Small-TM` is a small-footprint transaction with a single scatter zone value update. `Small-TM` resembles the case where a lock-prefixed instruction is used to enforce atomicity on a single variable.
- `Large-TM` is a large-footprint transaction where all scatter zones are updated for each zone. `Large-TM` resembles the update of multiple variables in one critical section.

Figure 2 show the results for 8 threads. Small Atomic (fine-grained lock) denotes the use of the OpenMP construction **#pragma omp atomic**. The X-axis denotes the number of scatters for each zone, and at each scatter count, the speedup is measured over the execution time of the corresponding serial version. For clarity, the graphs show only the results for the *SerControl* policy.

The results in the graph in Figure 2(a) lead to the following best-practice guideline: in a low-contention scenario it is preferable to convert multiple lock acquisitions or critical sections into a single transactional region, to better amortize the TM overhead, because the transaction footprint does not exhaust the available cache-line capacity. In high-contention scenarios, it is preferable to maintain transactions with smaller footprint, as evidenced by the graph in Figure 2(b).

Two additional best-practice guidelines arising from these results are: (i) *transaction footprint* and *contention* are the most important characteristics that dictate the TM performance on Intel's TSX as evidenced by the graphs on Figures 1(a) and 1(g). As shown in the graphs of Figures 1(e) and 1(f), one can reduce the transaction footprint by reducing the pollution in transaction or by increasing the temporal locality. These effects can be achieved by carefully designing the data structures of the application. (ii) Library-level support for TM matters. Moreover, try to use a serialization policy similar to the *SerControl* policy to avoid unnecessary serialization and to improve performance.

5. How does TSX perform with a more realistic benchmark?

This section uses the well-known, and widely used, STAMP [4] benchmark suite to evaluate the behaviour of real applications across the input-size dimension, similar to how Eigenbench and CLOMP-TM were evaluated previously. This study starts with the STAMP recommended configurations and data sets for use in real machines (for detailed information, see Table IV on [4]) and varies the input size to understand if real applications see the same effects seen in the synthetic benchmarks regarding working-set and transaction size. This is an unconventional use of the STAMP benchmarks, but it is very appropriate for this evaluation to understand the constraints that Intel's TSX creates for applications and application performance.

Figure 3 shows the speedup results of the *MaxRetry*, *Backoff* and *SerControl* policies, running on four threads, over the sequential execution in the same configuration.

For `Kmeans` the size of the transaction is proportional to the dimensionality of the space. Thus, we fixed the number of points (64 Kbytes), and vary the number of dimensions. `Kmeans` in low-contention scenario (the graph in Figure 3(a)) showed good performance results with the recommended parameters. `Kmeans` also exhibit high temporal locality and this further emphasizes the performance behaviour. However, it falls with increasing contention, as shown in the high-contention scenario (the graph in Figure 3(b)).

For `vacation`, we vary the number of records and, therefore the working-set size. `Vacation` (the graph in Figure 3(c)) showed a poor performance due to large-footprint transaction, the governing characteristic.

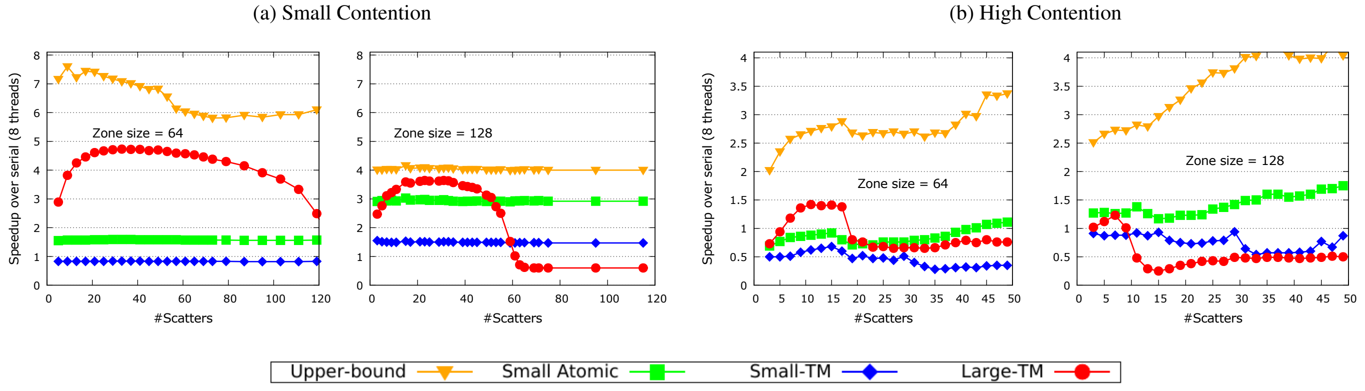
The `yada` (Yet Another Delaunay Application) benchmark implements Ruppert's algorithm for Delaunay mesh refinement. In the STAMP benchmark suite, `yada` does not come with sufficient datasets to test the behaviour of the application with different input sizes. We built a new dataset using the triangle application [15] through successive refinements from the data source `ladder` and we obtained data ranging from 32KB to 4MB on the number of input vertexes. `Yada` (the graph in Figure 3(d)) has relatively long transactions and a moderate amount of contention. With the increase of input vertexes, the transactions lead to large read- and write-sets and do not scale.

For `Genome` (the graphs in Figure 3(e)), we vary the number of gene segments. Even though the transactions in `genome` are of moderate length and have moderate read- and write-set sizes, the performance for `Genome` is very sensitive to increasing the number of segments — there is a dramatic performance drop because of longer transactions and larger working-set. The fraction of capacity-aborts per commit is lower for *SerControl* leading to slightly better performance. However, *SerControl*'s strategy is not effective to reduce the convoy effect for `genome`.

For `Intruder` (the graphs in Figure 3(f)), we vary the number of traffic flows. The limited performance comes from the large-footprint transaction and moderate-to-high levels of contention. The change in traffic flows modifies the working-set size, increasing the fraction of capacity aborts. The selective backing-off strategy of *SerControl* reduces the fraction of capacity aborts, as shown at the graph on the right, but is not sufficient to improve the performance when compared to *Backoff*
.

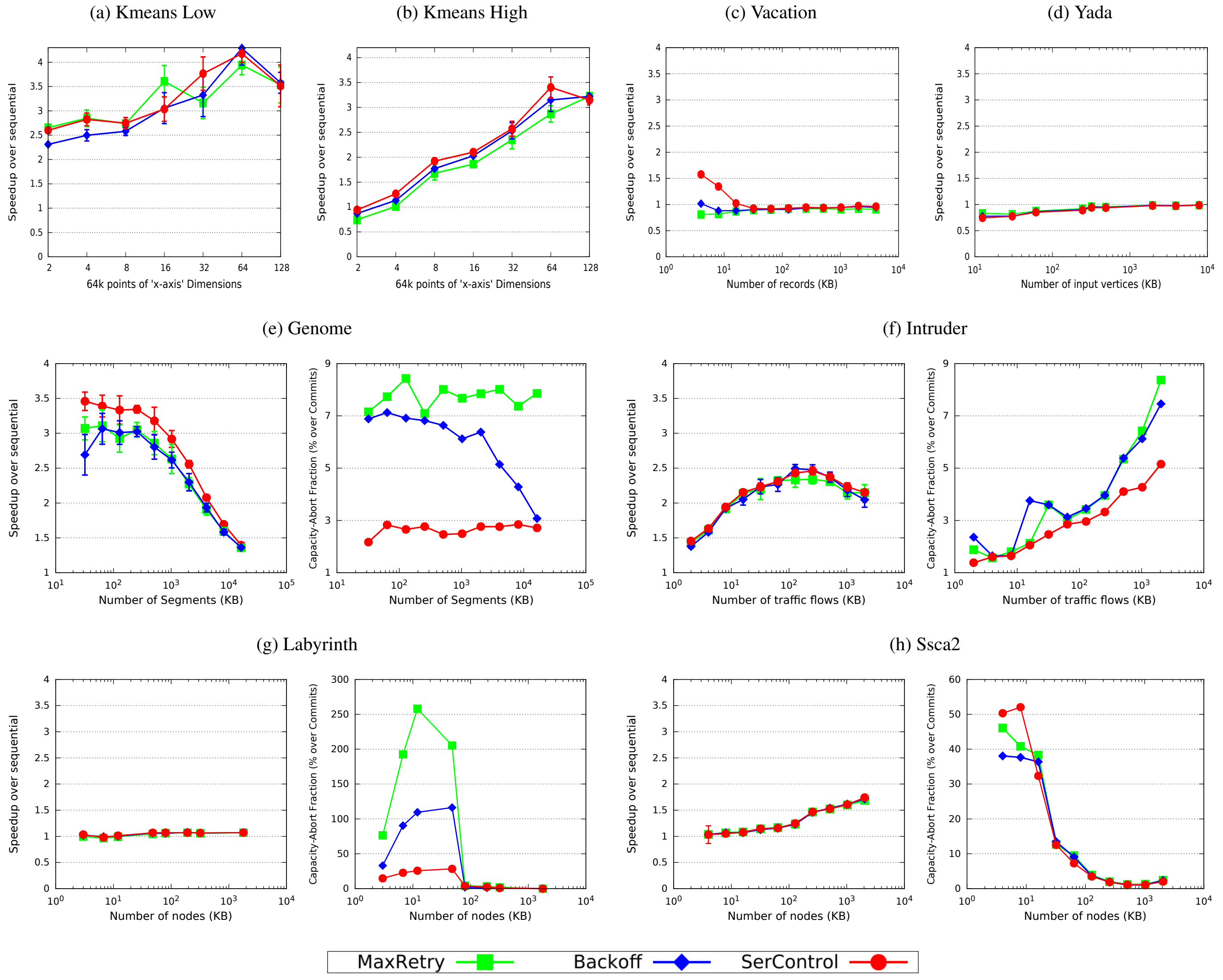
For `labyrinth` (the graphs in Figure 3(g)), we vary the number of dimensions that change both working-set size and transaction length. `Labyrinth` has very long transactions with very large read- and write-sets. The amount of contention is

Figure 2: Analysis of RTM using CLOMP-TM



(a) Small Contention (b) High Contention

Figure 3: STAMP applications – Analysis on Speedup and fraction of aborts to different input sizes.



(a) Kmeans Low (b) Kmeans High (c) Vacation (d) Yada (e) Genome (f) Intruder (g) Labyrinth (h) Ssca2

very high because of both the large number of transactional accesses to memory and the number of capacity-induced aborts, even for the *Backoff* policy. There is no performance gain in `labyrinth` for TSX with any of the policies.

For ssca2, we vary the number of nodes in the graph. Short transactions govern the performance of ssca2 (the left graph in Figure 3(h)). The increase in working-set size has little or no impact on application performance because only a small portion of time is spent in transactions. Also, differences in capacity-induced aborts over commits among the policies only appear for small working-set size (the right graph in Figure 3(h)).

The results in this section highlight the challenges faced by Intel's TSX on the STAMP benchmark suite. An analysis reveals that 70% to 80% of the aborts in STAMP are due to the architecture, not the application: (*e.g.*, page faults, system calls). In programs with short run times, these kinds of aborts appear to be have a dominant effect in the experiments, limiting the scalability of applications.

6. Tuning the Serialization Policies

Eigenbench was designed to study TM properties and thus do not account for changes in the parameters of the fallback policy used in the HTM system to provide forward-progress guarantee. However, an interesting question is whether the parameters in these policies affect performance. For instance, what should be the maximum number of retries in the *SerControl* policy used for the experiments described in Section 4.2? To address this question we built a tool, called htm-*pBuilder*, that acts as a wrapper over Eigenbench to allow independent exploration of the parameters of a given fallback policy. This tool takes as input a set of values for transaction properties and parameters and tailors fallback policies accordingly. The tool then automatically generates code for the various combinations of transaction properties and policy parameters, executes the code, and reports results. The following sections presents new insights, gained from the experiments performed with htm-*pBuilder*, into the tuning of the *SerControl* policy.

6.1. The effect of the Maximum Number of Retries

Section 4.2 described a systematic exploration of the characteristics of Intel's TSX. That exploration set the TM properties to a baseline configuration and then varied a single TM property at a time while allowing for a maximum of 20 retries in the *SerControl* policy. That threshold was obtained empirically as the best average result for a series of experiments with the RMS-TM [16] and STAMP benchmarks. Section 4.2 experiments are repeated in this section, but this time we use the htm-*pBuilder* tool to vary the maximum number of retries. These new experiments reveal that in applications with low-to-moderate contention a higher threshold value results in better performance while in applications with high contention this threshold should remain close to 20.

The experiments reported in Figure 4 support this observation. For Figure 4(a) the working-set is fixed at 32 Kbytes/thread and the write-ratio is 0.1 (pollution = 10%) — thus the contention level is low. The gradient plot reports the performance when the transaction length and the number of retries are varied. There is a significant performance change when the number of retries is greater than 20 for both small and large transactions. A comparison with Figure 1(a) on Section 4.2, where the number of retries was fix at 20, confirms that for that setting better performance is limited to transaction lengths between 30 and 60 words. In Figure 4(b) the transaction length is fixed at 50 words and the write-ratio is 0.1. The performance drop for 256 KBytes/thread when the maximum number of retries is equal 20, thus confirming the result in Figure 1(c) on Section 4.2. However the exploration with htm-*pBuilder* reveals that better performance can be obtained for larger work-sets if the maximum number of retries is increased.

A similar experiment, not reported here, explored the performance for different levels of contention when the maximum number of retries varies. This experiment revealed that, when the contention is high, increasing the number of retries does not improve performance to the same extent as in the low-contention case. This experimental result is most likely because of the limited data-cache line size in Intel's TSX. Thus, for high contention it is preferable to keep the retry threshold close to 20, or even less, to reduce the total number of aborts.

6.2. The Effect of Serialization on Capacity Aborts

To prevent frequent aborts because of capacity overflow, the *SerControl* policy serializes a transaction that has suffered two consecutive capacity aborts. The idea is that a capacity abort may be caused by competing transactions vying for the same storage resources. The htm-*pBuilder* facilitates an experimental evaluation of this strategy. Figure 4(c) shows the performance gradient when varying the transactional length and the number of consecutive capacity aborts allowed before the aborted transaction is serialized. The retry threshold utilized was 20, the working-set were 32 Kbytes/thread and the write ration was 10%. These transaction properties allow comparisons with the results reported in Section 4.2. The experiment revealed that the strategy was correct but, perhaps, adjusting this value to 3 or 4 could be a better solution.

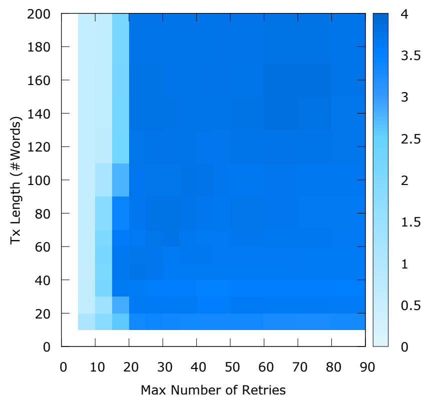
6.3. Applying the new findings to the STAMP benchmark suite

The experiments reported in Sections 6.1 and 6.2 used a set of synthetic programs generated by Eigenbench. A practical question is whether the finding that a higher value for the maximum number of retries can be beneficial applies to actual benchmarks. To answer this question, this section studies that finding in STAMP benchmarks.

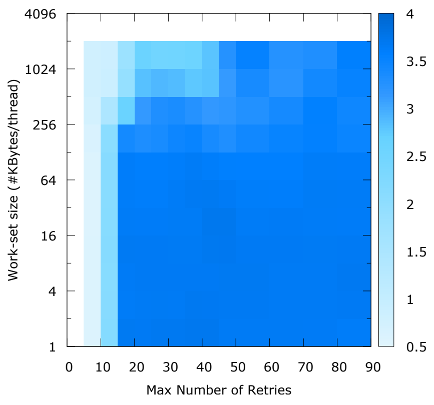
This study varied the maximum number of retries from 20 to 90 in all STAMP benchmarks. The only benchmark where a significant change in performance is observed is genome. The graph in Figure 3(e) in Section 5 indicates that the performance

Figure 4: Two-dimensional Analysis with Eigenbench and *SerControl*

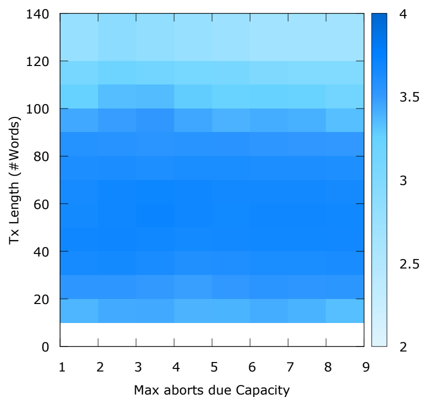
(a) Transaction Length X Retries (32 Kbytes/thread; write ratio = 0.1)



(b) Working-set size X Retries (50-word transactions; write ratio = 0.1)

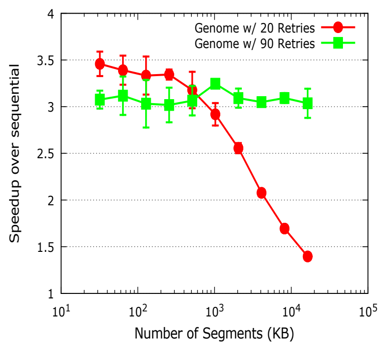


(c) Transaction Length X Capacity (32 Kbytes/thread; write ratio = 0.1; 20 retries)



of `genome` is sensitive to transaction length. Figure 5 compares the speedup, over sequential execution, for maximum number of retries of 20 and 90. A value of 90 yields the best performance, but there is no significant difference for values above 70. The shape on the graph for 90 retries show that, although the performance had a small drop for lowest number of segments, it maintains almost constant, regardless of the number of segments, thus confirming the observations in the graph on Figure 4(a).

Figure 5: Study of maximum retries with STAMP



6.4. *Avoiding the lemming effect*

The strategy adopted in the fallback policy is to retry the execution of the transaction — with or without a time delay — to attempt to complete the transaction execution speculatively. In the face of persisting failure, a transaction must be completed by running it in a non-speculative execution mode. A common solution is to acquire a global lock to prevent other transactions from committing concurrently. However, the acquisition of a lock by a transaction causes every other transaction to abort. This can cause a chain effect, also known as the *lemming effect*, where the aborted transactions also try to acquire the lock [17].

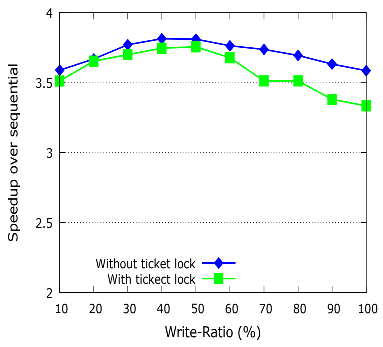
An alternative technique to the single global lock strategy is to use an auxiliary lock to prevent this lemming effect[18]. The idea is to guard the global lock acquisition with another lock. Aborted transactions have to acquire this auxiliary lock before serializing. This auxiliary lock is not added to the read set of transactions, thus avoiding the chain reaction effect.

In this experiment the auxiliary lock is a *ticket lock*. The *ticket lock* works as follows. Two memory locations - a *queue ticket* and a *dequeue ticket* — are accessed atomically. Initially both locations contain the value 0 to indicate that the ticket is not held. When a transaction needs to serialize its execution, it atomically reads and then increments the queue ticket. It then atomically compares the queue ticket that it read with the dequeue ticket's value. If they are the same, the transaction is permitted to try to obtain the global lock. If they are not the same, then another transaction must already be acquiring, or holds, the global lock and this transaction must busy wait or yield. When a transaction releases the global lock, it atomically increments the dequeue ticket thus allowing the next waiting transaction to acquire the global lock.

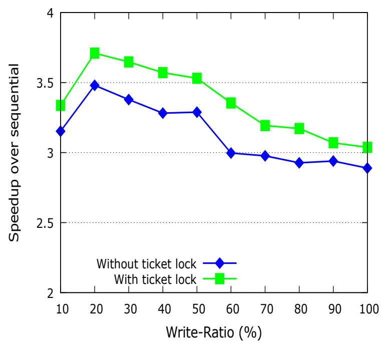
The experiment reported in Figure 6 used the *SerControl* policy with at most 20 retries. For a transaction length of 30 words the overhead of the use of the ticket lock resulted in a performance degradation. With a larger transaction length of 70 words the ticket lock results in a small performance improvement over the standard single-global-lock solution.

Figure 6: Study of the use of ticket lock to mitigate lemming effect using Eigenbench

(a) Transaction Length = 30 words (120 bytes)



(b) Transaction Length = 70 words (280 bytes)



6.5. *Reproducing the Performance Behaviour of Applications*

Applications usually have a mix of various transaction types and exhibit complex memory access patterns. One of the advantages of Eigenbench, that extends to htm-*pBuilder*, is the ability to mimic real applications by measuring the appropriate TM values and then mapping these values to eigen characteristics. These characteristics can be obtained via instrumentation-based profilling, simulation or analysis of the source code. However, htm-*pBuilder* has more expressive power than Eigenbench and allows developers to produce faster and more accurate assessments of the behaviour of HTM applications.

For instance, htm-*pBuilder* can be used to predict the performance of an application, originally synchronized using with locks, when it is re-written to use TM for synchronization. Htm-*pBuilder* can also help answer questions such as: (a) whether it is preferable to convert multiple lock acquisitions or critical sections into a single transactional region; or (b) if it is better amortize the overhead necessary to execute a transaction; or (c) whether it is preferable to maintain transactions with smaller footprint to reduce wasted work.

Htm-*pBuilder* can also be used to improve existing applications that use TM by enabling the evaluation of potential modifications without needing to prototype, thus reducing development time and cost. In our experiments, we used htm-*pBuilder* to emulate `genome` and `intruder` through instrumentation in fallback policy. These applications were executed with the tailored policy guided by scripts generated by htm-*pBuilder* to collect the size (measured in number of clock cycles) of dynamic transactions that commit. Clustering these results revealed the different types of static transactions. Then, new instrumentations of the fallback policy collected the write-ratio and temporal locality, counting the number and frequency of different locations (i.e. cache lines) touched by the transactions. With these results, we could emulate both applications and, from these emulations, we executed the programs to measure performance. Finally, we made some changes in the parameters (e.g., size of static transactions) to reflect hypothetical modifications and executed again the emulated applications to estimate the impact on performance due to the proposed modifications.

7. Related Work

Yoo et. al. in [5] presents an evaluation of Intel's TSX for High-Performance Computing. They claim that the first implementation of HTM in Haswell processors has significant performance potential. Using benchmarks and applications, they investigate some preliminary techniques to best utilize Intel's TSX, in such as lockset elision and transactional coarsening, but they do not provide library-level support for TM.[8]

Wang *et al.* presents an Intel's TSX performance characterization using a simple array-access microbenchmark [6]. They identify several important trends such as the relationships between transaction size, write ratio, retry count, transaction abort rate and performance. This study offers a more detailed study of the limits of TSX. It also introduces the *SerControl* policy that outperforms both a simple MaxRetry and the backing-off policy.

Diegues and Romano introduce a method to automatically tune the number of attempts to reschedule a transaction in Intel's TSX [19]. Their method is motivated by the observation that no single configuration of the software fallback policy can perform efficiently in every workload and application. Their policy, Tuner, uses Upper Confidence Bounds (UCB) to select a strategy to adjust the number of retries given to a transaction when a capacity abort occurs. The goal is to try to distinguish between transient capacity failures caused by temporary cache addressing issues and persistent capacity failures caused by excessive amounts of required speculative state. The double-check in *SerControl* is a simple heuristic that attempts to achieve the same effect. Diegues and Romano used an exploration technique similar to a hill climbing/gradient descent

[8]We are currently integrating the proposed *SerControl* policy within the `libitm`, the TM library of the GCC compiler.

search to tune the number of attempts, a problem not explored in this work.

Calciu *et al.* present a novel hybrid transactional memory (HyTM), called Invyswell, that uses hardware transactions from Haswell's RTM in conjunction with software transactions from a heavily modified design of InvalSTM [22], an STM designed to provide scalability and performance for large transactions with notable contention [20]. As our results show, Haswell's RTM performs best for small transactions with low contention, as it imposes no instrumentation overhead in fallback policies. On the other hand, InvalSTM performs best for large transactions with high contention, because it can make highly informed contention management decisions through its commit-time invalidation process. Although Calciu *et al.* show that Invswell performance compares favourably to most STMs and HTMs, this kind of hybrid solution is beyond the scope of the proposed work.

8. Conclusion

Transactional memory is a natural fit for multi-core architectures, but the success of transactional memory will be partially determined by the quality of early implementations. This evaluation shows that the best-effort transactional memory provided by Intel's Haswell is simple and capable of improving performance over a variety of workloads. However, performance can depend strongly on the software support systems. While transaction footprint and working-set size constraints dictate the range of effective transactions, choices made in the lock fallback policy can considerably affect performance, especially when capacity-limited transactions are executed. The dynamic nature of the cache means that the capacity-aborted signal is not a completely reliable indicator that a transaction will not complete. This observation is supported by the success of the *SerControl* fallback policy, which allows transactions that suffer capacity aborts to retry. A unique evaluation of transaction footprint and working-set size through input modification of the STAMP benchmarks confirms that the best-effort nature and capacity limitations of Intel's HTM underscores that TM is not a catch-all solution to parallel synchronization.

Acknowledgment

The authors would like to thank the anonymous reviewers for the insightful comments. This work is supported under grant 2013/08293-7, São Paulo Research Foundation (FAPESP) and by the Natural Sciences and Engineering Research Council of Canada (NSERC).

[1] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, K. Olukotun, Eigenbench: A simple exploration tool for orthogonal TM characteristics, in: The IEEE International Symposium on Workload Characterization (IISWC), 2010, pp. 1–11.

[2] M. Schindewolf, B. Bihari, J. Gyllenhaal, M. Schulz, A. Wang, W. Karl, What scientific applications can benefit from hardware transactional memory?, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, Los Alamitos, CA, USA, 2012, pp. 90:1–90:11.

[3] Intel Corporation, Chapter 12: Intel's Transactional Synchronization Extensions (TSX) Recommendations, `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html` (Jul. 2013).

[4] C. C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford Transactional Applications for Multi-Processing, in: The IEEE International Symposium on Workload Characterization (IISWC), Seattle, WA, USA, 2008, pp. 35–46.

[5] R. M. Yoo, C. J. Hughes, K. Lai, R. Rajwar, Performance evaluation of intel&reg; transactional synchronization extensions for high-performance computing, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, New York, NY, USA, 2013, pp. 19:1–19:11.

[6] M. D. Wang, M. Burcea, L. Li, S. Sharifymoghaddam, G. Steffan, C. Amza, Exploring the performance and programmability design space of hardware transactional memory, in: The ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), Raleigh, NC, USA, 2014.

[7] M. Herlihy, J. E. B. Moss, Transactional memory: architectural support for lock-free data structures, in: International Symposium on Computer Architecture (ISCA), ACM, San Diego, California, USA, 1993, pp. 289–300.

[8] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, D. A. Wood, LogTM: Log-based transactional memory, in: International Symposium on High-Performance Computer Architecture, IEEE Computer Society, Austin, Texas, USA, 2006, pp. 254–265.

[9] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, D. Grossman, ASF: AMD64 extension for lock-free data structures and transactional memory, in: International Symposium on Microarchitecture (MICRO), Atlanta, Georgia, USA, 2010, pp. 39–50.

[10] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, K. Olukotun, Transactional memory coherence and consistency, in: International Symposium on Computer Architecture (ISCA), IEEE Computer Society, Mnchen, Germany, 2004, pp. 102–113.

[11] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, D. A. Wood, LogTM-SE: Decoupling hardware transactional memory from caches, in: International Symposium on High-Performance Computer Architecture, Phoenix, Arizona, USA, 2007, pp. 261–272.

[12] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, M. Michael, Evaluation of blue gene/q hardware support for transactional memories, in: International Conference on Parallel Architectures and Compilation Techniques, PACT'12, ACM, Minneapolis, Minnesota, USA, 2012, pp. 127–136.

[13] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, D. A. Wood, Performance pathologies in hardware transactional memory, in: International Symposium on Computer Architecture (ISCA), 2007, pp. 81–91.

[14] L. Ceze, J. Tuck, J. Torrellas, C. Cascaval, Bulk disambiguation of speculative threads in multiprocessors, in: International Symposium on Computer Architecture (ISCA), IEEE Computer Society, 2006, pp. 227–238.

[15] J. R. Shewchuk, Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator, in: M. C. Lin, D. Manocha (Eds.), Applied Computational Geometry: Towards Geometric Engineering, Vol. 1148 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 203–222.

[16] G. Kestor, S. Stipic, O. Unsal, A. Cristal, M. Valero, RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications, in: The ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), Salt Lake City, Utah, USA, 2009.

[17] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir,

K. Moore, D. Nussbaum, Applications of the adaptive transactional memory test platform, in: TRANSACT '08: 3rd Workshop on Transactional Computing, 2008.

[18] Y. Afek, A. Levy, A. Morrison, Programming with hardware lock elision, in: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, ACM, New York, NY, USA, 2013, pp. 295–296. doi:10.1145/2442516.2442552.

[19] N. Diegues, P. Romano, Self-tuning intel transactional synchronization extensions, in: 11th International Conference on Autonomic Computing (ICAC 14), USENIX Association, Philadelphia, PA, 2014.

[20] Calciu, Irina and Gottschlich, Justin and Shpeisman, Tatiana and Pokam, Gilles and Herlihy, Maurice, Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory, in Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, 2014, pp. 187–200.

[21] 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. 17931807, 2010.

[22] J. E. Gottschlich, M. Vachharajani, and J. G. Siek. An efficient Software Transactional Memory using commit-time invalidation. In Proceedings of the International Symposium on Code Generation and Optimization (CGO), April 2010.