Exploring the Performance and Programmability Design Space of Hardware Transactional Memory

Mike Dai Wang
U. of Toronto
dai.wang@mail.utoronto.ca

Mihai Burcea
U. of Toronto
burceam@eecg.toronto.edu

Linghan Li
U. of Toronto
linghan.li@mail.utoronto.ca

Sahel Sharifymoghaddam
U. of Toronto
sahel.sharifi@gmail.com

Greg Steffan
U. of Toronto
steffan@eecg.toronto.edu

Cristiana Amza
U. of Toronto
amza@eecg.toronto.edu

Abstract

In this paper, we study the programmability and performance design space of the new hardware transactional memory (HTM) framework provided by Intel's Haswell architecture. Towards this, we first present an Intel TSX performance characterization using a simple array access microbenchmark. Through a comprehensive study we identify several important trends, such as, the relationships between, transaction size, write ratio inside transactions, retry count, and transaction abort rate and performance.

Next, we explore code transformations such as, computation splitting and privatization, for optimizing the performance of Moldyn, a molecular dynamics simulation from the CHARMM [4] molecular dynamics simulation and analysis package. We leverage our TSX performance characterization to guide and minimize our parametrization efforts for our Moldyn code transformations.

We found that a hardware TM solution using computation splitting and privatization can be both easier to program and also outperform a hand-tuned fine-grain pthread locks solution including those same optimizations.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: parallel programming; D.3.3 [*Language Constructs and Features*]: concurrent programming structures; D.4.1 [*Operating Systems, Process Management*]: concurrency, mutual exclusion

Keywords Transactional Memory, Speculative Execution

1. Introduction

Transactional Memory (TM) [6, 15, 16] is a promising parallel programming paradigm for exploiting the increasing parallelism available in chip multiprocessors. It is aimed to provide correctness, performance and improved programmability over traditional lock-based synchronization approaches. Despite some recently reported successes with moderately complex realistic applications such as CAD tools like VPR [2, 3], games [15] or synchronization within an operating system [17], TM adoption is not yet widespread. This is partially due to the significant amounts of overhead associated with Software Transactional Memory (STM) systems. However, with the recent trend of increased Hardware Transactional Memory (HTM) support in commercial systems - e.g., in IBM's BGQ [11, 19] and in Intel's Transactional Synchronization Extensions (TSX) in the Haswell architecture [21] - research focus on uses of TM for exploiting parallelism is now more relevant than ever before.

With the newly introduced TSX extensions, speculative execution of transactions can be done in hardware, and locks protecting critical sections can be elided whenever possible. This has the potential of greatly improving application performance, by executing critical sections optimistically, rather than through unnecessary locking. However, HTM is no magic bullet, and brings its own constraints in the form of limitations imposed by the capacity of the associated hardware resources. In this paper, we explore the design space for opportunities towards a best of all worlds design, where the performance advantages of HTM could be coupled with programmability and flexibility advantages similar to the STM.

Towards this, we first use off-line profiling on a simple microbenchmark to extensively explore the performance sensitivity of the HTM to a variety of application patterns and HTM configuration parameters. Our profiling pass identifies the most important parameters and the most likely guidelines for reducing our programming and configuration effort for other applications. Based on the profiling results, we design a set of optimizations that can be manually applied to an application. We plan to extend the investigation presented in this paper to implement semi-automated support for i) leveraging existing profiling data to narrow down the range of the most beneficial parameter settings corresponding to specific application patterns and ii) applying the manual optimizations we introduce in a generic way. In the following, we present our investigation in more detail.

We use an array access microbenchmark which can be extensively parametrized for HTM performance characterization and profiling purposes. The performance model derived from our profiling shows that HTM performance depends on several factors, including transaction size, retry count, the write ratio of the application, the randomness in the access pattern of the application and the flavor of Intel HTM (HLE or RTM).

We find that the HTM performance is most sensitive to the transaction size, and the number of retries the HTM is configured with.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Transact '14, March 2, 2014, Salt Lake City, UT, USA.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Moreover, the functional dependency of performance to each of the above factors, is non-linear, thus making our profiling pass necessary towards guiding programming/configuration effort for regular applications. For example, performance depends on transaction size in a mountain-like function: the optimum solution is not encountered at either end of the spectrum, but on an in-between plateau. Even more of interest is that there is a significant jump in performance at a certain point with increasing the retry count.

We use the profiling results, and the associated performance models for finding the best performance/programmability sweet spot with the HTM when programming a molecular dynamics simulation, called Moldyn, from the CHARMM [4] molecular dynamics simulation and analysis package. We design and manually code the following code transformations: i) computation splitting: chopping coarse grained transactions into smaller transactional units that can still be executed in parallel and ii) data and computation privatization: privatizing certain data and moving parts of computations outside of transactions.

We leverage the performance models derived with the array micro-benchmark to drive our parameter selection e.g., for the size of transactions and number of retries. This allows us to avoid running all possibilities in order to fine tune Moldyn, when using our code transformations.

For both benchmarks, we targeted programmability and ease of use by utilizing a previous STM system developed locally, libTM [6, 15, 16] in order to integrate its simple to use API with Intel TSX. The libTM API allows application developers to easily create threads, parallelize functions and set up transactional begin and end scopes. Thus, tedious implementation details, such as, thread creation, thread pooling, and implicit barrier creation are all transparent to the programmer. We extend this API to include Intel TSX support and fallback locking. This API is then leveraged in both benchmarks we used in this study with ease. Further fine tuning and manual optimizations were done in addition to the underlying libTM features and will be automated in our upcoming work.

We present results for our Intel TSX performance characterization using the array access microbenchmark and the associated trends contributing to performance improvements and abort rate reductions. We also present results for our performance enhancing code transformations with Moldyn. All experiments were run on a PC with 4 2-way Hyperthreaded Intel i7-4770 cores that supports Intel TSX. The solution space for Moldyn which we investigated consists of software TM, hardware TM using Intel TSX, regular and optimized Pthread locking versions. The best solution we found through our investigation is a manually tuned HTM solution that uses lock elision with a global-lock fallback path. The optimized HTM solution remarkably combines both better performance and is easier to parallelize compared to our most fine-tuned solution of the same application using Pthread locks. In Moldyn, using 8 threads we observed a speedup of 2.76x compared to a sequential baseline version. We further observe that the hand-tuned HTM solution provides similar parallelism with a fine-grained locking solution without incurring the increased memory footprint of maintaining the fine grained locks. Hence, the HTM provides a better performing solution (by approximately 10%) with less programming effort than fine-grained locking.

The remainder of this paper is organized as follows: Section 2 provides background information. Section 3 describes experiment setup. Section 4 describes our case study on characterizing TSX. Section 5 discusses the results and optimizations when using TSX to evaluate a realistic application: Moldyn. Section 6 provides analysis and discussion on the impact of optimizations. Section 7 looks at related work in this field. Section 8 provides a glimpse of our upcoming work and finally Section 9 concludes our findings.

2. Background

2.1 Intel Transactional Synchronizations Extensions

Intel Transactional Synchronizations Extensions (Intel TSX) [13, 21] are a recent addition to the Intel architecture that provide programmers with a way to leverage the support for hardware transactional memory offered by the Haswell architecture. Intel TSX comes in two basic flavors: the first, called Hardware Lock Elision (HLE), is intended to help programmers to benefit from the hardware transactional support for already-existing applications that employ lock-based synchronization. HLE makes available prefixes to existing instructions that allow the hardware to first attempt execution by speculatively eliding locks in the code; if speculation fails, execution of the speculative section is restarted, with the locks actually being acquired this time.

The second flavor of hardware transactional support is called Restricted Transactional Memory (RTM). RTM provides an interface allowing programmers to specify a region of code that will be executed transactionally. In the case of an abort, the transaction can either be retried in hardware or fallback to a separately defined software path.

In addition to the instruction extensions, hardware transactions supported by TSX utilize the CPU cores' L1 caches and the underlying cache coherence protocol for conflict detection. Transactions are tracked at cache line granularity (64 bytes on our experiment platform).

Further details on Intel TSX implementation and analysis are available in the references cited above.

In this study, we focus our efforts on RTM to take advantage of the flexibility in defining a custom software fallback path and the opportunity for possible automated optimizations in the future.

2.2 LibTM STM Library

libTM [14, 15] is a highly-customizable STM library written in C++ with an intuitive API. A programmer can use libTM to parallelize an application by following three steps: first, data structures that will be used inside transactions must be converted to libTM-specific data types (tm_types); libTM tracks conflicts by using operator overloading for these tm_types.

Second, the infrastructure of parallel threads can easily be created and managed; libTM offers a convenient way to manage a thread pool and create implicit barriers through the following macros:

- CREATE_TM_THREADS (num_threads);
- DESTROY_TM_THREADS (num_threads);
- PARALLEL_EXECUTE (num_threads, parallel_func, arg);

In the PARALLEL_EXECUTE macro, the $parallel_func$ is a function that will be executed in parallel (which may contain multiple transactions, or call other functions that may contain transactions), and takes arg and the id of the thread executing it as arguments.

Finally, BEGIN_TRANSACTION() and END_TRANSACTION() constructs let the programmer create a scope to enclose the specific portion of code that should be executed transactionally.

The underlying libTM STM implementation supports a variety of conflict detection and conflict resolution policies to allow for maximum flexibility.

2.3 LibTM TSX Support

To take advantage of the performance benefits provided by Intel TSX while maintaining programmability, libTM has been modified to support TSX / RTM as its hardware transactional memory component. Specifically:

- The libTM API remains unchanged and transparent to ensure the same programmability and minimal programming effort across benchmarks.
- To evaluate TSX / RTM fairly and free from the influence of STM overhead, we completely detach libTM's STM execution core and the associated statistics tools.
- XBEGIN, XEND, XTEST instructions and various other tools provided by the Intel TSX / RTM instruction set extensions are utilized to execute transactions in hardware as opposed to using the STM. We note that the flexibility of altering conflict detection and conflict resolution policies is no longer within the control of the programmer nor libTM.
- RTM alone does not provide any progress guarantees for an application: a hardware transaction may keep retrying (and aborting) indefinitely. To ensure progress, we implement a spinlock-based software fallback path for every RTM transaction started. TSX will speculatively attempt to elide the spin locks but should any transactions abort, critical sections will be protected by acquiring the spin locks in software. Our implementation supports both multiple fine-grained spin locks and one global spin lock as the software fallback path.
- In order to facilitate transaction retries, users can provide a parameter to specify and adjust the number of retries in RTM before falling back to software execution. The retry parameter combined with the hardware status registers [1] allow libTM to maximize the likelihood of transactions committing successfully.

3. Experimental Setup

Our experiments are conducted on a desktop PC with 4 2-way Hyperthreaded Intel i7-4770 Cores at 3.4GHz and 12GB of RAM, 4x32KB L1 caches, 4x256KB L2 caches and a shared 8MB L3 cache. All benchmarks are written in C++ and compiled with GCC 4.8.1 to run on a 64 bit Ubuntu Linux with 3.5.0-40 kernel. Threads are bound to processors ensuring each thread run exclusively on its (Hyperthreaded) core. All results collected represent an average over 3 runs.

4. Case Study: Array Access Microbenchmark

In this section, we provide detailed descriptions of how we characterize TSX performance using a simple microbenchmark and discuss the trends we observe based on the collected results.

4.1 Microbenchmark Overview

To evaluate and characterize HTM, we conduct a case study using a synthetic array access microbenchmark. As shown in Figure 1, the benchmark maintains a one dimensional integer array where its elements are accessed through either reads or writes based on pre-generated access patterns. The microbenchmark accepts 3 tuning parameters: $num_threads$, $write_ratio$ (percentage of writes inside a transaction), and txn_size (number of array accesses inside a transaction). The generated access patterns can be in one of two access "modes" to model typical application behaviour: "random" mode is self-explanatory; in "contiguous" mode, an offset in the array is randomly chosen for each transaction, and the transaction will access the subsequent txn_size elements starting from that offset. The order of reads and writes is also randomly chosen but based on $write_ratio$.

For the purpose of this study, we set the main integer array to 3000 elements and the problem size to 5 million iterations. Through varying the tuning parameters, we aim to identify possible trends and effects of each parameter on TM-related metrics. Specifically

```
ArrayAccessBenchmark() {
  GenerateAccessPattern();

  for every iteration in N iterations {
    BEGIN_TRANSACTION()
    AccessArray() //read/write based on access pattern
    END_TRANSACTION()
  }
}
```

Figure 1. Pseudo code for Array Access Microbenchmark

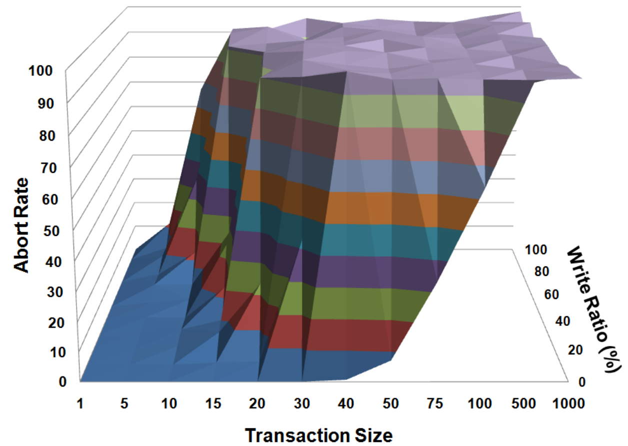

Figure 2. Abort rate vs. txn size vs. write ratio for the array access microbenchmark with random accesses, 4 threads, and 0 transaction retries.

we measured: abort rate (percentage of transactions that did not commit out of the total number of transactions started), transaction throughput (number of shared data accesses per second), and transaction retry count.

4.2 TSX Characterization Results

Next we present a representative subset of our characterization results.

Figure 2 shows a 3 dimensional plot of abort rates vs. transaction size vs. write ratio for random access patterns. For read-dominated transactions, the abort rate becomes very high when transaction size approaches 20 accesses. For write-dominated transactions, transaction sizes larger than 10 accesses experience a sharp rise in abort rates. We observe a similar trend with contiguous access patterns, but with slightly higher transaction size ceilings due to less cache thrashing.

Figure 3 shows a simplified snapshot of array access throughput plotted against transaction size and write ratio for random access patterns. We observe that a "plateau" or performance peak exists at $write_ratio$ of 10 and txn_size of 40 resulting in a throughput of approximately 5700 shared accesses per second. Our findings show that overall performance does not necessarily correlate with the smallest transaction size and likely is application-specific.

Finally, Figure 4 shows that for certain transaction sizes (100 accesses in this particular case), restarting hardware transactions 4 or 5 times significantly reduces abort rate by up to 80%. Repeating the experiment with random access pattern produced similar results.

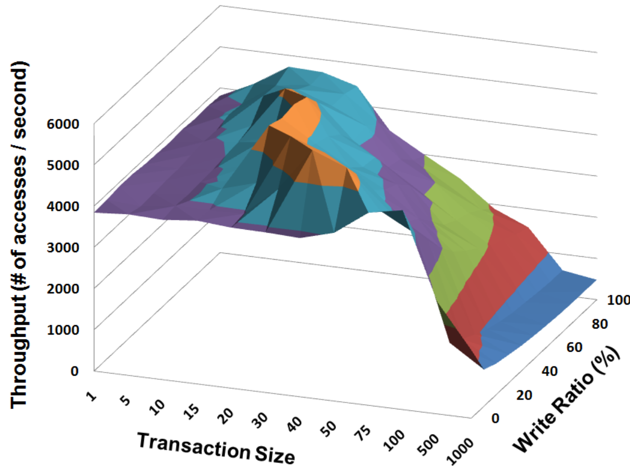

Figure 3. Throughput vs. txn size vs. write ratio for the array access microbenchmark with random accesses, 4 threads, and 0 transaction retries.

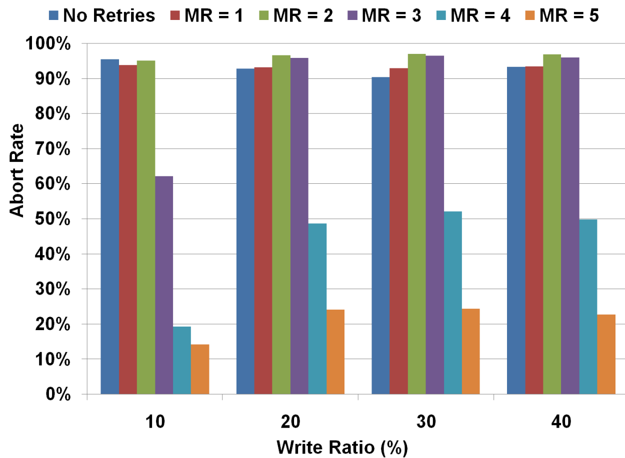

Figure 4. Abort rate vs. write ratio and the number of retries for the array access microbenchmark with contiguous accesses, 4 threads.

4.3 TSX Performance Trends

With these results in mind, we summarize 5 trends contributing to better TSX performance and lower transaction abort rates. They are listed below:

1. Transaction size directly correlates with abort rates and smaller transactions exhibit higher likelihood of committing.
2. Low abort rate does not always guarantee to higher overall application performance. Rather, a "sweet spot" of performance likely exists for applications using Intel TSX.
3. Contiguous access consistently outperforms random access when comparing abort rates. Such relationship directly relates to the utilization of cache locality and the amount of cache thrashing due to the access pattern.
4. Retrying aborted hardware transactions multiple times has the potential of drastically reducing abort rate.
5. For certain parameter combinations, HLE outperforms RTM in terms of transaction abort rates. However, investigation on HLE performance is still ongoing at the time of writing and the focus of this study will be on RTM with reasons explained in Section 2.1.

5. Evaluation on Realistic Benchmark: Moldyn

In this section, we evaluate the performance impact of TSX on a more realistic application: Moldyn. Detailed description of different versions of Moldyn and the details of our optimizations will be presented. Specifically we aim to translate the trends identified from the case study in Section 4 into code transformation and optimization techniques. We will present evaluation results comparing different versions of Moldyn in terms of runtime, scalability, and abort rate.

Moldyn is a molecular dynamics simulation in 3 dimensional space that iterates over a number of time steps. It computes interaction forces between neighboring molecules in the $ComputeForces$ stage, then updates molecule coordinates and velocities based on the previously computed forces in the $Update$ stage. The interacting neighbors list of every molecule is recomputed in $BuildNeighbors$ after a fixed number of time steps. Finally, the overall system energy and velocities are tracked and logged after each time step to ensure application correctness. Figure 5 shows a simplified sequential version of how Moldyn works.

```
Moldyn() {
  InitializeMolecules();
  for every time step in N timesteps {
    every Kth time step {
      UpdateNeighbours();
    }
    ComputForces();
    Update();
  }
}
```

Figure 5. Simplified Pseudo code for Moldyn.

In this evaluation we model a 3D space containing 32,000 molecules with randomly generated, uniformly distributed initial positions and the simulation lasts 30 time steps.

5.1 Evaluation Baselines

The original sequential version(**seq**) of Moldyn is used as a starting point, where all molecule force calculations and energy updates are done with 1 single thread. Next, the sequential version is parallelized using POSIX threads (Pthreads) but with 3 different synchronization mechanisms: Locks, STM, and HTM. We list the corresponding parallelized baselines accordingly below:

- Pthread mutex locks and explicit barriers are used to create coarsed-grained lock based **lock.coarse** and fine-grained lock based **lock.fine**.
- The STM system used in the evaluation is libTM as described in Section 2.2. As STM specific evaluation and optimizations are beyond the scope of this study, we chose the best performing version (**libtm.stm**) as a comparison reference. **libtm.stm** uses "fully optimistic" conflict detection and "abort readers" conflict resolution policies.
- There are also 2 HTM baselines using the libTM library with TSX support as described in Section 2.3. First is a naive version of Moldyn with extremely coarse transactions (**tsx.coarse**) containing entire computation loops of all molecules. Such coarse transactions can potentially result in large amounts of

cache conflicts leading to aborts. Second, a version with transactions containing the smallest possible independent computations(**tsx.fine**) accessing only 1 molecule is evaluated.

5.2 Optimizing Moldyn

In this section, we provide details on how specific code transformations and optimizations are translated from the case study trends to Moldyn. As a result, we create 2 optimized code versions in **lock.opt** and **tsx.opt**.

Through profiling and parallelizing Moldyn, we identify several critical sections within the code base. Using the function $ComputeForces$ as an example, we list 4 categories of code transformations and optimizations below. Optimizations 1 and 2 apply to both **lock.opt** and **tsx.opt**, whereas optimizations 3 and 4 apply more directly to **tsx.opt**.

1. **Data Privatization**: As with any parallelization effort, privatization of shared data can lead to less contention and smaller critical sections. Shared global variables used to accumulate system-wide measurements (total energy, etc) are privatized in this case.
2. **Computation Privatization**: Corresponding to data privatization, CPU intensive calculations such as finding 3 dimensional displacement between molecules are moved outside of critical sections and executed in parallel.
3. **Computation Splitting & Merging**: Transactions that are too large or too small could result in high abort rates and possibly low performance. **tsx.opt** tries to find the "sweet spot" by computing each pair of molecule updates inside a single transactions as opposed to thousands of molecules or only a single molecule. This logical transaction size allows **tsx.opt** to maximize RTM performance without the danger of compromising atomicity.
4. **Multiple Transaction Retries**: Aborted transactions in **tsx.opt** have the opportunity to retry committing in hardware multiple times instead of immediately taking the software fallback path.

As a result of the aforementioned optimizations, we show the pseudocode for both **lock.opt** in Figure 6 and **tsx.opt** in Figure 7.

```
ComputeForces() {
  for each pair i, j assigned to thread {
    Read molecule positions into temporaries
    Compute 3D displacement
    Calculate force delta due to interaction

    // fine-grained locks
    lock (molecule i)
    Update molecule i 3D force vector
    unlock (molecule i)
    lock (molecule j)
    Update molecule j 3D force vector
    unlock (molecule j)

    // data privatization
    Accumulate per thread energy totals
  }
  lock (total energy)
  Accumulate system wide total energy
  unlock (total energy)
}
```

Figure 6. Pseudo code for the optimized lock based version of $ComputeForces$ in Moldyn.

```
ComputeForces() {
  for each pair i, j assigned to thread {
    // computation privatization
    Read molecule positions into temp.
    Compute 3D displacement
    Calculate force delta due to interaction

    // computation splitting & merging
    BEGIN_TRANSACTION()
    Update molecule i 3D force vector
    Update molecule j 3D force vector
    END_TRANSACTION()

    // data privatization
    Accumulate per thread energy totals
  }
  BEGIN_TRANSACTION()
  Accumulate system wide total energy
  END_TRANSACTION()
}
```

Figure 7. Pseudo code for the optimized TSX version of $ComputeForces$ in Moldyn.

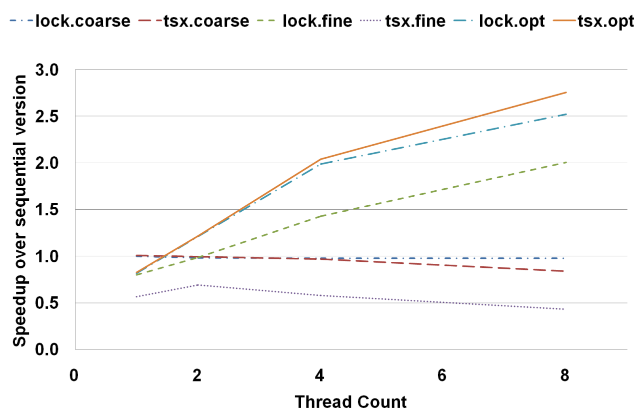

Figure 8. Scaling results for Moldyn. The Y axis shows normalized speedup over the sequential version as a baseline.

5.3 Code Transformation Results

Next, we compare the results of optimized Moldyn against baseline versions.

Figure 11 shows speedup comparisons of different versions of Moldyn in this experiment. In the best case, using 8 threads we show **tsx.opt** achieves up to 2.76x speedup compared to the baseline version and even beats the performance of **lock.opt** by approximately 10%.

Figure 8 shows the scalability factors of all versions of Moldyn as we increase the number of threads. We observe good scalability for **tsx.opt**, outperforming all other versions.

Table 1 shows the abort rate statistics of the TSX/RTM versions. We observe a significant reduction in both transaction count and abort rates with **tsx.opt**.

Figures 9 and 10 show the effects of increasing RTM transaction retry count. In the 8 thread case of Moldyn, allowing aborted transactions to retry up to 4 times can produce approximately 6.5x speedup versus if the aborted transactions take the fallback path right away (i.e., no retries). This also greatly reduces the abort rate

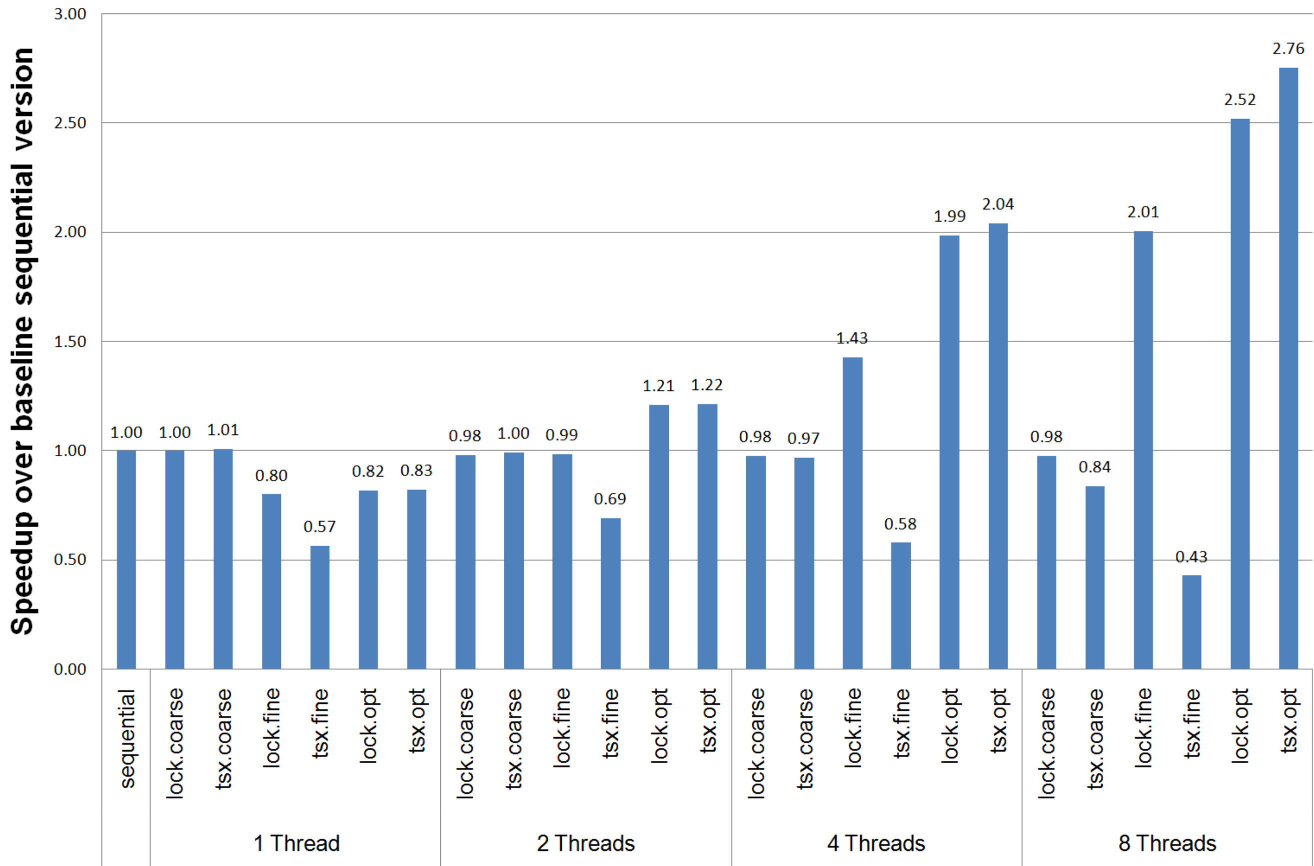

Figure 11. Speedup comparison for Moldyn, normalized with respect to the baseline sequential version.

Version	1 thread		2 threads		4 threads		8 threads	
	txn count	abort rate	txn count	abort rate	txn count	abort rate	txn count	abort rate
tsx.coarse	62	52%	124	52%	248	52%	496	52%
tsx.fine	211M	0%	211M	11%	211M	73%	211M	86%
tsx.opt	75M	0%	75M	0%	77M	2%	87M	13%

Table 1. Transaction count and abort rates for Moldyn using Intel TSX/RTM.

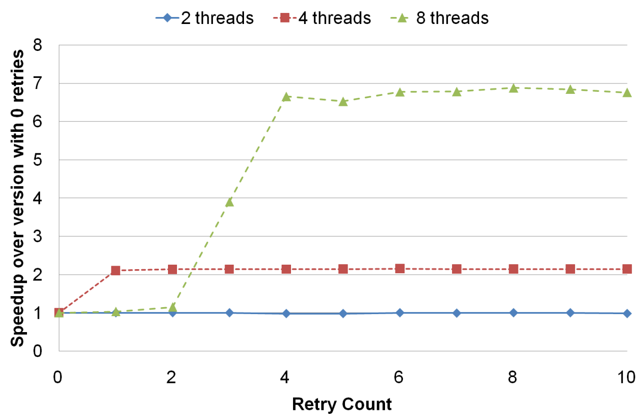

Figure 9. Speedup vs number of retries for Moldyn. The Y axis shows normalized speedup over the version with no retries as a baseline.

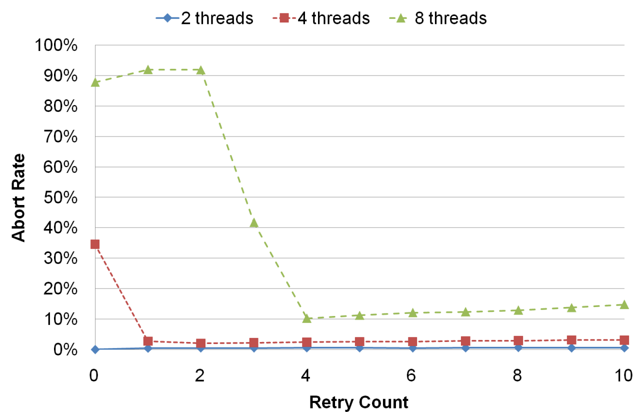

Figure 10. Abort rates vs number of retries for Moldyn.

of the total transactions from approximately 90% to approximately 10%.

Further analysing the presented results, we show that with both **lock.coarse** and **tsx.coarse**, there is very little performance improvement. **tsx.coarse** tracks the performance of **lock.coarse** simply due to all transactions aborting and taking the fallback path turning the **tsx.coarse** version into a coarse-grained locked version. These results are not surprising due to the contentious nature of Moldyn stages and the naively implemented coarse-grained critical sections. This shows that more than naive efforts are needed to achieve meaningful speedup.

Next we examine a set of fine-grained versions with both locks and RTM transactions (Figure 11). While the **lock.fine** version eventually achieves a speedup with 8 threads, the large amounts of locking, unlocking and the locks themselves can be a significant source of overhead. The **tsx.fine** version does not perform well and results in slowdowns in all cases. We further discuss the reasoning in Section 6.

To limit the scope of this study, we only present the best performing STM implementation of Moldyn: **libtm.stm**. Due to high STM overhead, **libtm.stm** results in a slowdown of 6.3x with 8 threads compared to the baseline. However, we do note that the scalability of the STM system trends positively as we increase the number of threads as shown in Figure 8.

6. Analysis and Discussion

In the following section, we analyze in detail the obtain results and provide reasoning behind our applied optimizations. Our goal of this analysis and the subsequent discussions is to determine if and how these TSX related optimizations can be applied in a more generic fashion towards other applications.

6.1 Privatization

Privatization is an often used technique when parallelizing applications, and Moldyn is no exception.

In Moldyn, most critical sections are both large and computationally intensive. They perform time-consuming calculations such as finding out the three-dimensional displacements between interacting molecules, and computing changes in kinetic energy due to velocity changes. By applying computation privatization, critical sections are reduced to smaller sizes and more relevant calculations.

In addition to computationally intensive code sections, we also observe heavy data contention. Our experiments using the Intel Performance Counter Monitor (PCM) tool indicate that approximately 75% of RTM transaction aborts are due to memory contention (experiments not shown for brevity). The sources of contention mostly stem from frequent reads and updates of shared variables. These include major data structures such as arrays tracking molecule forces, velocities, positions, interacting partners and accumulated kinetic energy and temperature counters. By privatizing counters, modifying shared data structures, and padding molecule objects for better cache line alignment, we reduce the amount of overall data contention significantly.

As indicated in Figure 11, performance improvements between **lock.fine** and **lock.opt** versions are mainly due to privatization. And in the case of TSX based versions, privatization also plays a major role in producing a meaningful speedup. This leads us to believe that applying privatization appropriately can produce better performance in most applications.

6.2 Optimal Transaction Size

Section 4.2 shows that transaction size is an important factor to RTM transaction commit success rate. Examining abort details for RTM transactions in **tsx.coarse**, we observe that large transactions (containing multiple iterations of computation or update loops) make up almost the entirety of the 52% aborted transactions (Table 1). It is apparent that the large size of these transactions is stressing the L1 cache resources, leading to repeated aborts.

On the opposite end of the spectrum in **tsx.fine**, the abort rate decreases if we map the fine-grained locks to small transactional regions containing 1 iteration of only 1 molecule update; but having very small transactions leads to a drastic increase in the number of total transactions: from 248 to 211 million. Although fine-grained locks reduce false sharing, fine-grained transactions may worsen performance due to the cache line size tracking granularity of RTM.

These results are consistent with the ones obtained with the array access microbenchmark shown in Figure 3, where a low abort rate does not necessarily lead to better performance (similarly due to the overhead caused by a large number of transactions).

The next natural step on the optimization path is to find the "sweet spot" of transaction size, abort rate and overall performance. Guided by the TSX characterization results from our case study, we did not need to exhaustively search for all possible transaction sizes in Moldyn. Instead we take advantage of modified molecule objects from privatization, which are padded to align with cache line sizes. We close in on the "sweet spot" by splitting large transactions into computing 1 pair of interacting molecules. Calculating molecule pairs in a transaction not only ensures atomicity, but the size change in **tsx.opt** also results in performance improvements as shown in Section 5.3.

We do note however, that such "sweet spots" in transaction size are application-specific and are not trivial to find. Computation splitting also needs to be done with considerable care to provide atomicity guarantees. Our characterization provides a guideline but finer tuning is needed if maximum performance gains are desired. Perhaps profiling with appropriate tools can provide additional guidance in such searches.

6.3 RTM Transaction Retries

Similar to our findings for the array access microbenchmark, tuning the number of retries for the RTM hardware transactions can provide significant performance improvements for Moldyn. As shown in Figures 9 and 10, a sharp "jump" exists in performance improvements as the number of transaction retries increases. The results however, indicate that finding out where this "jump" likely depends on application characteristics. Rather than a fixed optimum retry limit number like suggested by Yoo et. al. [21], one might obtain better results by further tuning.

This application-specific parameter may be utilized in on-line profiling tools. At the time of writing, experiments are still ongoing to further examine the impact of retry count on specific Moldyn stages.

6.4 Fallback Lock Granularity

The software fallback path is an important part of our TSX investigation as described in Section 2.3; however, it may represent another potential source of performance degradation, depending on the locking scheme employed.

As we have explained, Moldyn critical sections are long and computationally intensive. This makes it all the more likely that, if we use a single global lock for all fallback paths, unrelated critical sections will abort each other unnecessarily. We can reduce lock contention by replacing the global fallback lock with multiple fine-grained locks, but this is non-trivial, as we must ensure atomicity of reads and writes to shared variables in various parts of the critical section. Furthermore, as we pointed out earlier, these additional data structures used for locking will put extra pressure on the

already-strained cache resources, and can become another cause of aborts.

Another option is to assign locks to a particular group of shared data, but this will come at the expense of significant programming effort and finesse.

At the time of writing, we have conducted limited experiments in reducing the granularity of fallback locks and have not observed any significant performance improvements. While it is possible that fine-grained fallback locks may provide additional benefits in certain scenarios, programming effort needed in identifying the specific locks needed for different transactional sections may compromise programmability. We see using a global fallback but with split transactions into related logical segments to be a promising approach given the opportunity of automation.

7. Related Work

Transactional Memory (TM) has been the subject of extensive studies in the last decade or so [8–10, 16, 18], but traditionally the focus has been been either on entirely software approaches, or on hybrid implementations with the hardware part being simulated [7, 20]. As actual hardware transactional support has become available only recently from Intel [12] and IBM [11], new work has been looking at how to successfully couple the existing hardware support with software approaches.

Wang et. al. [19] also investigated IBM's support for hardware TM; they look at how the STAMP benchmark suite [5] performs on the BlueGene/Q, and classified several categories of applications in terms of their suitability to use both TM in general, and the BG/Q flavour of hardware TM more specifically.

Probably the closest-related work to ours is the very recent study by Yoo et. al. [21], where the authors look at the performance of Intel's Transactional Synchronization Extensions (Intel TSX) on several benchmark suites, and also investigate some preliminary optimization techniques to improve the compatibility of code with Intel's TSX support. While their work certainly outdistances ours in breadth through the sheer number of applications investigated, we argue that our own study complements their work, by going into more depth regarding certain performance aspects. For example, we had the freedom to do an exhaustive investigation of the effect of several parameters on application performance with our array microbenchmark (whereas such flexibility is not available while evaluating standardized benchmark suites). Furthermore, where Yoo et. al. give little insight into the importance of retries, we show the significant impact that choosing the right number of retries has. Another insight that our work provides is that, perhaps unintuitively, the number (or ratio) of aborts is not necessarily directly correlated with the best execution time; however, we acknowledge that it is possible this behaviour does not occur on specific classes of applications or transactional code.

8. Future Work

8.1 Automatic Optimizations with Hybrid libTM

While many of the optimizations described in this study are manually applied, with the already existing libTM API, our goal is to design and implement support for some of our optimizations in a generic way, that is applicable in a wide variety of cases. Specifically, we see opportunities to apply the techniques used in this study towards an automated hybrid TM system. libTM currently supports overloading operators of transactionally accessed variables in its underlying STM system. We plan to further extend this interface along with per thread private buffers to reduce the length of critical regions. This form of automatic privatization also logically split and reduce transaction sizes, thus allowing further possible improvements. We plan to use such ideas and use RTM to perform only tightly packed shared accesses. By nesting RTM transactions to perform read validations and write flushes inside the STM commit stage can potentially achieve similar performance improvements we see in our current results. Finally, transaction retry is an addition parameters that can be a subject of a more comprehensive study and potentially create opportunities for dynamically optimizing its usage.

8.2 Cross-Phase Parallelization Optimizations

We are currently investigating an additional code transformation optimization to further improve parallelism in barrier-based programs, such as, Moldyn. We note that barriers, as a synchronization mechanism, can be overly restrictive and unnecessarily delay threads from execution, even in cases where threads could actually proceed with their computation without waiting for slower threads. Towards optimizing parallelism while respecting ordering constraints, we plan to modify our existing libTM framework to include awareness and enforcing of transaction ordering, through user-provided transaction identifiers, and committing transactions in identifier order. This would create opportunities for barrier removal hence additional parallelism in the application.

9. Conclusions

In this study, we explore the trends and trade-offs in performance and programmability for hardware transactional memory, particularly Intel Transactional Synchronization Extensions. We systematically investigate the solution space of software TM, hardware TM using Intel TSX, regular and optimized Pthread locking versions, for a balanced solution between programmability and performance.

Using an array micro-benchmark, we derive performance models for a range of application patterns and parameter settings. We further explore code transformations and optimizations for improving performance, including computation privatization, transaction splitting or other size reduction.

We show that, with our code transformations and optimizations, guided by our parameterized models for TSX, a speedup of up to 2.76x is achieved versus the baseline sequential version. Our optimized TSX version of Moldyn involved lower programming effort while at the same time outperforming (by approximately 10%) the best hand optimized fine grain Pthread version. We thus show that high performance is achievable with a programming interface that is the familiar STM interface, without the need to use complicated fine-grained locking.

We are planning to explore techniques for leveraging our performance models, parameter tuning trends, and code transformations by both on-line and off-line profilers or automated code optimization tools.

In the long run, we envision a natural progression in TM research leading to some form of hybrid TM system where the performance advantages of HTM would be married to the programmability and flexibility of STM. This combination would greatly help programmers towards achieving the ultimate desirable for the programming experience: correctness, simplicity, and performance.

References

[1] *Intel TSX enabling and optimization recommendations*. https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[2] *Versatile Place and Route*. http://www.eecg.toronto.edu/vpr.

[3] S. Birk, J. Steffan, and J. Anderson. Parallelizing FPGA placement using Transactional Memory. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 61–69, 2010. .

[4] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular

energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983. ISSN 1096-987X. . URL http://dx.doi.org/10.1002/jcc.540040211.

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[6] H. Chafi, J. Casper, B. Carlstrom, A. McDonald, C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 97–108, 2007. .

[7] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0. . URL http://doi.acm.org/10.1145/1168857.1168900.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *DISC*, 2006.

[9] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching Transactional Memory. *SIGPLAN Not.*, 44:155–165, 2009. ISSN 0362-1340. URL http://doi.acm.org/10.1145/1543135.1542494.

[10] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-based Software Transactional Memory. In *PPoPP*, 2008.

[11] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim. The ibm blue gene/q compute chip. *Micro, IEEE*, 32(2):48–60, 2012. ISSN 0272-1732. .

[12] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*. http://software.intel.com/sites/default/files/69/60/41604.

[13] D. Kanter. *Analysis of Haswell's Transactional Memory*. http://www.realworldtech.com/haswell-tm/1/.

[14] D. Lupei. A Study of Conflict Detection in Software Transactional Memory. In *Master thesis*, 2009. URL https://tspace.library.utoronto.ca/bitstream/1807/18804/3/Lupei_Daniel_200911_MASc_thesis.pdf.

[15] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. Amza. Transactional Memory Support for Scalable and Transparent Parallelization of Multiplayer Games. In *EuroSys*, 2010.

[16] A. McDonald, J. Chung, H. Chafi, C. C. Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on Chip-Multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 63–74, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2429-X. . URL http://dx.doi.org/10.1109/PACT.2005.11.

[17] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 87–102, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. . URL http://doi.acm.org/10.1145/1294261.1294271.

[18] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *SPAA*, 2008.

[19] 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*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. . URL http://doi.acm.org/10.1145/2370816.2370836.

[20] L. Yen, J. Bobba, M. Marty, K. Moore, H. Volos, M. Hill, M. Swift, and D. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 261–272, 2007. .

[21] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. . URL http://doi.acm.org/10.1145/2503210.2503232.