A Study of the Potential of Intel’s Transactional Synchronization Extensions for Hybrid Transactional Memory

by

Mike Dai Wang

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2014 by Mike Dai Wang
Abstract

A Study of the Potential of Intel's Transactional Synchronization Extensions for Hybrid Transactional Memory

Mike Dai Wang
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2014

Hardware Transactional Memory (TM) attempts to deliver on the promises made with Software Transactional Memory by enabling scalable parallel execution with low programming effort. Intel’s TSX is a recent such effort to provide hardware support for Transactional Memory on commodity hardware. This dissertation studies the limitations of TSX and proposes both manual and automated techniques for improving application parallelism. We find that using only TSX may result in high abort rates and poor performance for certain workloads due to hardware resource limitations. Using the Moldyn benchmark, we present a series of manual optimizations on TSX to achieve performance comparable to fine-grained locking. We then translate the identified manual optimizations into automated techniques in the form of a software-hardware hybrid transactional memory library called HyTM. Using a realistic benchmark, SynQuake, we experimentally show that HyTM can significantly increase the fraction of transactions that take advantage of TSX and execute in hardware.
Acknowledgements

First and foremost, I would like to offer my gratitude and appreciation to my supervisor, Professor Cristiana Amza. Her knowledge, guidance, vision, and patience were of tremendous help to me throughout my graduate school experience. It was an honor and privilege to be supervised by Professor Amza.

I would also like to thank my examination committee members: Professor Ben Liang, Professor Ashvin Goel, and Professor David Lie for their time, comments and valuable feedback.

Throughout the writing of this dissertation and my research journey, I have learned a great deal from my colleagues and lab mates. I thank them for sharing their knowledge, especially Mihai Burcea, whose expertise were instrumental to overcoming many technical challenges.

Last but not least, I would like to thank my family for their love and support, especially my wife Cathy, who stood by my side with encouragements and understanding throughout the last two years.
## Contents

1 Introduction .................................................. 1

2 Background .................................................... 6

  2.1 Transactional Memory Concepts .......................... 6
      2.1.1 Speculative Execution ................................. 6
      2.1.2 Shared Access Tracking ............................... 7
      2.1.3 Conflict Detection .................................... 7
      2.1.4 Conflict Resolution ................................... 8

  2.2 Intel Transactional Synchronizations Extensions  ...... 9
      2.2.1 HLE vs. RTM ............................................ 9
      2.2.2 Contention Management ............................... 10
      2.2.3 Progress Guarantee and Software Fall-back ........... 10
      2.2.4 TSX Abort Causes ..................................... 11

3 TSX-Based Library Design .................................. 12

  3.1 Hardware TM Platform: Intel Haswell .................. 12
  3.2 Application Programming Interface ...................... 13
  3.3 Software Fall-back Path Implementation ................. 14

4 TSX Modelling and Characterization ....................... 16

  4.1 Case Study 1: Array Access Micro-benchmark .......... 16
      4.1.1 TSX Modelling Results ............................... 17
      4.1.2 TSX Performance Trends .............................. 20

  4.2 Case Study 2: Moldyn, A Molecular Dynamics Simulation Benchmark 21
      4.2.1 Evaluation Baselines ................................ 22
      4.2.2 Optimizing Moldyn ................................... 23
      4.2.3 Moldyn Results ........................................ 25
      4.2.4 Analysis of Optimizations ........................... 27
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>TSX with simple software global lock fall-back (FB) path.</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>Abort rate vs. txn size vs. write ratio for the array access micro-benchmark using 4 threads, 0 transaction retries, and random access pattern.</td>
<td>17</td>
</tr>
<tr>
<td>4.2</td>
<td>Abort rate vs. txn size vs. write ratio for the array access micro-benchmark using 4 threads, 0 transaction retries, and contiguous access pattern.</td>
<td>18</td>
</tr>
<tr>
<td>4.3</td>
<td>Abort rate vs. txn size vs. write ratio for the array access micro-benchmark using 4 threads, 0 transaction retries, and random access pattern.</td>
<td>19</td>
</tr>
<tr>
<td>4.4</td>
<td>Throughput vs. txn size vs. write ratio for the array access micro-benchmark using 4 threads, 0 transaction retries, and contiguous access pattern.</td>
<td>19</td>
</tr>
<tr>
<td>4.5</td>
<td>Abort rate vs. write ratio and the number of retries for the array access micro-benchmark with contiguous accesses, 4 threads.</td>
<td>20</td>
</tr>
<tr>
<td>4.6</td>
<td>Speedup comparison for Moldyn, normalized with respect to the baseline sequential version.</td>
<td>25</td>
</tr>
<tr>
<td>4.7</td>
<td>Moldyn speedup comparison when using different number of threads. The Y axis shows normalized speedup over the sequential version as a baseline.</td>
<td>26</td>
</tr>
<tr>
<td>4.8</td>
<td>Speedup vs number of retries for Moldyn. The Y axis shows normalized speedup over the version with no retries as a baseline.</td>
<td>26</td>
</tr>
<tr>
<td>4.9</td>
<td>Abort rates vs number of retries for Moldyn.</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>tm_types created from wrapped objects.</td>
<td>33</td>
</tr>
<tr>
<td>5.2</td>
<td>Multiple tm_type wrapped application objects sharing the same version tag, which acts similar to a lock.</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>Hybrid TM transaction flow diagram.</td>
<td>37</td>
</tr>
<tr>
<td>5.4</td>
<td>Transaction and read-set size reduction using version tags in the hybrid layer with hardware commit.</td>
<td>39</td>
</tr>
<tr>
<td>6.1</td>
<td>SynQuake game map data structure [34].</td>
<td>46</td>
</tr>
</tbody>
</table>
6.2 SynQuake client processing stages .......................... 47
6.3 SynQuake player distribution with 1 center quest. ............. 51
6.4 SynQuake player distribution with 4 quests in each of the quadrants on the game map. ........................................... 51
6.5 SynQuake player distribution where players are moving around randomly with no quest. .................................... 52
6.6 SynQuake transaction composition with 1 quest at the center of the game map. .................................................. 54
6.7 SynQuake transaction composition with 4 quests in each of the quadrants on the game map. ........................................... 55
6.8 SynQuake transaction composition where players moving around randomly with no quests. ............................. 55
6.9 Abort cause breakdown of all transactional versions of SynQuake entering the hardware-only layer running a game with 1 quest at the center of the game map. ........................................... 57
6.10 Abort cause breakdown of all transactional versions of SynQuake entering the hardware-only layer running a game with 4 quests in each of the quadrants on the game map. ........................................... 58
6.11 Abort cause breakdown of all transactional versions of SynQuake entering the hardware-only layer running a game where players moving around randomly with no quests. ..................... 58
6.12 Abort cause breakdown of all transactional versions of SynQuake entering the hybrid layer with hardware commit phase running a game with 3 different quest scenarios. ......................... 60
6.13 Breakdown of abort causes of all transactional versions of SynQuake entering the hybrid layer with software commit phase running a game with 3 different quest scenarios. ......................... 61
6.14 Execution time comparison between all transactional versions of SynQuake running a game with 500 players, 10000 cycles, and under 3 different types of game scenarios. ......................... 62
6.15 Scalability comparison of all transactional versions of SynQuake against single thread execution running a game with 1 quest at the center of the game map. ........................................... 63
6.16 Scalability comparison of all transactional versions of SynQuake against single thread execution running a game with 4 quests in each of the quadrants on the game map. ........................................... 64
6.17 Scalability comparison of all transactional versions of SynQuake against single thread execution running a game where players moving around randomly with no quests. ............................. 64

6.18 hytm.opt transaction composition with 1 quest at the center of the game map. ................................................................. 66

6.19 hytm.opt transaction composition with 4 quests in each of the quadrants on the game map. ......................................................... 66

6.20 hytm.opt transaction composition where players moving around randomly with no quests. ................................................................. 67

6.21 Execution time of hytm.opt running a game with 500 players, 10000 cycles, and under 3 different types of game scenarios. ......................... 67
Chapter 1

Introduction

As the number of cores in modern processors increases, the demand for software applications to be scalable and high performance is also growing. Despite efforts towards creating generic techniques, development of parallel applications is typically perceived as complex, error-prone, time consuming or a combination of the three. Traditionally, lock-based synchronization has been used for non-embarrassingly parallel applications. With coarse-grained locks, the parallelization effort can be minimal but their use commonly results in poor performance. Using fine-grained locks has the potential of greatly improving performance and scalability, but is error-prone and requires considerable programming effort. Developers in both academia and industry have been examining alternative concurrency control mechanisms providing programmability, performance, and scalability.

Transactional Memory (TM) [8, 25, 30] is a promising parallel programming paradigm for exploiting the increasing parallelism available in chip multiprocessors and it has been proposed as a viable alternative to traditional lock-based synchronization techniques. TM aims to provide correctness, performance and to improve programmability over traditional lock-based approaches.

With transactional memory, a programmer can invoke a transaction in a multi-threaded application and rely on the TM system to make its execution appear atomic in a serializable fashion [18]. TM systems seek higher performance by speculatively executing transactions concurrently in a lock-free manner and only committing transactions that do not encounter conflicting accesses. Transactions can commit concurrently if there are no conflicting accesses, but they incur the overhead of access tracking, aborting and rolling-back when multiple accesses occur on the same word, block, or object, where at least one access is a write [26].

Despite some recently reported successes with moderately complex realistic applications, such as, CAD tools like VPR [3, 4], games [25] or synchronization within an
operating system [33], TM adoption is not yet widespread and the uptake on the idea is rather slow. 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 [38, 16] and in Intel’s Transactional Synchronization Extensions (TSX) in the recently released Haswell architecture [42] - research focus on uses of TM for exploiting parallelism is now more relevant than ever before.

In this dissertation, we place the focus of our HTM study on Intel’s TSX, due to its commercial availability and low cost of ownership. It is our hope to not only understand TSX behaviour, but also study the potential of using HTM systems, such as, TSX for software-hardware hybrid TM implementations. Intel’s TSX HTM implementation did not require a full hardware redesign of the existing CPU hierarchy. In fact, it relies largely on existing components to perform speculative execution of transactions, and a technique called lock elision is used to run critical sections in parallel without locking whenever possible [9].

It is the hope of TSX system designers that parallel applications can take advantage of speculative execution modes and better exploit available parallelism. Specifically, for applications that have already been heavily optimized with small critical sections, utilizing TSX has the potential of greatly improving performance, by executing critical sections optimistically, rather than through unnecessary locking. While, in theory, HTM can be of great assistance in creating scalable applications, it alone is no magic bullet. Implementations such as TSX bring their own constraints such as hardware resource capacity limitations. These constraints can become glaring if applications are not well optimized or developers are not familiar with the optimization strategies.

In this dissertation, we focus our efforts on investigation of two aspects of TSX: modelling as well as profiling TSX behaviour, and evaluating optimizations for applications using TSX in both manual and automatic approaches. Towards this, we first use off-line profiling on a simple micro-benchmark 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 the applications under study. Based on the profiling results, we create a set of HTM models and detect limitations that would be the basis for progress guarantee strategies and as tuning points for optimization. Such models will present a set of approaches that application developers can utilize in order to create well performing and efficient parallel programs. We verify the identified models on a medium-sized benchmark, Moldyn, with measured performance
improvements and scaling potential. Once we identify the manual TSX optimization parameters, we shift our focus towards creating a hybrid TM system called HyTM. Through using a set of established APIs and the incorporation of software TM components, HyTM attempts to ease the programming effort while providing levels of performance comparable to the manually optimized baselines using locks. Finally, HyTM is evaluated using a complex benchmark, SynQuake, and compared with the most optimized lock based implementations.

In the following, we present our investigation in more detail.

We use an array access micro-benchmark 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 flavour 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. 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. Other factors that affect performance are the write ratio of the application, the randomness in the access pattern of the application, and the choice between TSX flavours.

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 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.

Our investigations reveal that a manually tuned HTM solution that uses lock elision
with a global-lock fall-back path can achieve comparable performance to a heavily opti-
mized version of the same benchmark using fine-grained Pthread locks. As an example, in
Moldyn, using 8 threads and the sequential version as a baseline, we observed a speedup
of 2.76x compared to 2.52x achieved by the Pthread lock based version. We further
observe that the hand-tuned HTM solution provides similar parallelism to a fine-grained
locking solution without incurring the increased memory footprint of maintaining the
fine grained locks.

With well understood TSX behaviour models, we design and implement a hybrid TM
library: HyTM. The library will be a mixture of TSX instructions and a set of interfaces
aimed at providing programmability, based on a previous STM system developed locally,
libTM \cite{8,25,30}. The libTM API allows application developers to easily create threads,
parallelize functions and set up transactional begin and end scopes. Thus, tedious imple-
mentation 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
in addition to leveraging and replacing the original STM core with our TSX based core
to complete the hybrid solution. We investigate all the potential optimizations on HyTM
for their effectiveness and we further extend our hybrid TM implementation by allowing
applications to optionally adjust the conflict detection granularity.

In addition to the simple benchmarks we use for evaluation, different versions of our
system will also be pitched against each other using a realistic application: SynQuake.
SynQuake is the server component of an open-source port of the popular game, Quake,
where flocking behaviour of massive number of players presents an interesting challenge
and usage scenario for TM evaluation. We study the effectiveness of our system by
investigating the entire solution space consisting of hardware TM using Intel TSX, HyTM,
regular and optimized Pthread locking versions. SynQuake was chosen to evaluate our
hybrid TM for several important reasons: it is a more realistic and complex benchmark
than existing benchmark suites; an earlier evaluation of SynQuake with an STM showed
potential for a TM-based approach versus locking; and finally SynQuake has gaged the
attention of several TM groups who show significant interest in the benchmark source
code. In a context where TM benchmarks are scarce, SynQuake has the potential to
become a de-facto standard in TM benchmarking.

In summary, this dissertation makes the following contributions:

1. Characterize and model Intel’s Transactional Synchronization Extensions for better
   understanding of HTM properties, and limitations.

2. Produce a summary of guidelines and recommendations for manual tuning of high
performance parallel applications using TSX.

3. Design and implement an original Hybrid TM library providing developers ease of programming and high performing parallel applications.

4. Implement and showcase several benchmarks relevant to HTM and Hybrid TM research including the array access micro-benchmark, and Moldyn.

5. Study HTM and Hybrid TM performance on a realistic application: SynQuake.

The remainder of this paper is organized as follows: Chapter 2 provides background information. Chapter 3 describes a basic TSX-based library design. Chapter 4 provides results and identified trends from our TSX modelling and characterization. Chapter 5 details our implementation of the hybrid TM library and its core features. Chapter 6 describes details of SynQuake benchmark and discusses experimental results from the SynQuake benchmark. Chapter 7 looks at related work in this field and finally Chapter 8 concludes our findings and provides a glimpse of our upcoming work.
Chapter 2

Background

2.1 Transactional Memory Concepts

Researchers in the past several decades have explored a wide variety of STM and HTM implementation options and system details. In this section, we present the generic concepts and details most relevant to our ongoing research. The background we present next will focus on shared access tracking and contention management in preparation for more specific topics in the latter part of this dissertation.

2.1.1 Speculative Execution

The fundamental difference between TM and lock-based synchronization lies with the premise of speculative execution and how atomicity is guaranteed.

Lock-based synchronization is blocking and pessimistic in nature; whether it is mutexes, semaphores, or conditional variables, critical sections are "locked" regardless of whether different executing threads truly contend for it or not. Such unnecessary locking creates the effect of false sharing. As a result of false sharing, the potential for greater parallel execution or scalability is limited by the size of the critical section.

On the contrary, TM attempts to resolve that limitation by executing critical sections in transactions speculatively. These transactions will execute the critical sections in parallel based on the optimistic assumption that contention might not occur. The changes a transaction executing thread makes to shared memory locations will only become visible to other threads if the transaction successfully commits. The visible changes will commit atomically and appear to other threads as a single operation. If the transactions fails to commit, then it can abort and attempt to retry until successful. Theoretically using TM can greatly improve application performance and scalability especially if the number of
contentious accesses is small.

2.1.2 Shared Access Tracking

In order for any TM system to atomically write changes on commit and roll-back changes on abort, shared accesses (reads and writes to shared memory locations) must be tracked. Doing so allows the underlying TM system to perform synchronization in a correct manner. These tracked accesses are almost always done with a per-transaction write-set and read-set. Access tracking can be done in two very different ways: **Write-Buffering (WB)** and **Undo-Logging (UL)**.

1. **Write-Buffering (WB)** is also known as Multi-Versioning, where every transaction-executing thread does not write directly to shared memory locations but writes to a private write buffer/set instead. At commit time, if the transaction has not been aborted, the thread will “flush” its write buffer and actually commit to the shared memory locations in 1 atomic operation. If a transaction aborts, the executing thread simply discards its write buffer and rolls back to the beginning of the transaction to retry.

2. In **Undo-Logging (UL)**, all transactions commit their writes directly to shared memory locations, but they are required to keep a "log" in their write-set of all the old values they replaced. Upon successful commit, threads do not need to perform further actions since all the shared updates are already completed. In the case of aborts however, threads will need to "restore" the shared memory locations they have written to back to their old logged values.

Both of the described shared access tracking methods are used extensively in different TM implementations. They typically perform well in certain scenarios and carries overhead in different forms.

2.1.3 Conflict Detection

Depending on the shared access tracking mechanism, the TM system may deploy different types of Conflict Detection (CD) policies based on usage scenarios or conditions. CD represents a set of policies used by the TM systems to detect when conflicting accesses to shared memory locations occur. Typically, conflicts occur as a result of one transaction
writing to a shared memory location while another transaction accesses the same memory locations. There are two common types of conflict detection:

1. **Pessimistic CD** [29] acknowledges conflicts eagerly at encounter-time where the TM system is signaled immediately after a dependency violation has occurred. This allows the TM system to quickly respond and resolve the conflict right away. It is more prevalent in write-buffering TM systems to avoid wasting CPU cycles if transactions are destined to abort.

2. **Optimistic CD** [17] acknowledges conflicts lazily at commit-time. An aborted transaction can potentially re-use the contents in its cache from the previous failed commit attempt.

A TM system’s CD policies need to be able to distinguish and detect different types of data hazards in order to guarantee correctness and overall transaction serializability. The types of data hazards that could occur between reader and writer transactions are listed below:

- **Read After Write (RAW)**: a true dependency exists when a reader reads from a shared memory location after a writer has written to it (the memory location is tracked in the writer’s write set). RAW hazards occur when the writer commits before the reader.

- **Write After Read (WAR)**: an anti-dependency exists when a writer writes to a shared memory location that is tracked in a reader’s read-set. If the reader commits before the writer, there is no hazard as the writer has not made its changes visible to the reader. If the reader commits after the writer however, a WAR hazard occurs.

- **Write After Write (WAW)**: an output dependency exists when a writer writes to a shared memory location that is tracked in another writer’s write-set. If the application has no requirement on the transaction ordering and the writes have no other dependencies, then a WAW hazard is avoided. But most likely the writes depend on other earlier reads in their respective transactions that could form other dependency relationships.

### 2.1.4 Conflict Resolution

After conflicts are detected by the TM system, Conflict Resolution (CR) policies are needed to resolve the conflicts and determine if and when transactions need to be aborted. These CR policies typically focus on giving priority to either readers or writers:
1. **Wait-for-readers**: doing so gives priority to readers and writers must wait for reader transactions to finish (successfully or not) before attempting to commit. Doing so allows the TM system to resolve certain WAR and RAW hazards without having to abort the writer transaction. Such a policy is aimed at application scenarios where the number of reads outnumber the number of writes inside a transaction [27].

2. **Abort-readers**: gives priority to writers where all conflicting readers are immediately aborted and restarted. Doing so allows updates to shared memory locations to be of the highest priority and available to all threads immediately. Anything transactions performing reads will need to re-read the values to ensure the possession of the most up-to-date values. Abort-readers are typically implemented in conjunction with write-buffering TM systems and pessimistic CD [35].

Although CR appears to be simple, it is worth noting that aborting and restarting transactions can potentially lead to unexpected deadlock and live-lock scenarios. Considerable care usually is required to ensure the TM system can eventually make progress and complete execution.

As we have noted, TM contention management is a major topic of research, there are countless other CD and CR policies as well as variant of shared access tracking methods. Actual TM implementations typically deploy a mixture of the variants to serve a specific purpose or to conform to a specific system/standard.

### 2.2 Intel Transactional Synchronizations Extensions

Intel Transactional Synchronizations Extensions (Intel TSX) [42 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.

#### 2.2.1 HLE vs. RTM

Intel TSX comes in two basic flavours: 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 and commit changes in a single atomic fashion;
if speculation fails, execution of the speculative section is restarted, with the locks actually being acquired this time. XACQUIRE and XRELEASE prefixes can be used to specify the begin and the end of a transactional region.

The second flavour 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 fall back to a separately defined software fall-back path. The intrinsic prefixes used to specify begin and end of a transactional region are: XBEGIN and XEND. An additional instruction XABORT is available should application developers feel the need to explicitly abort a RTM transaction.

In this study, we focus our efforts on RTM to take advantage of the flexibility in defining a custom software fall-back path and the opportunity for possible automated optimizations in the future. We will be using TSX and RTM interchangeably throughout this section as well as the rest of the dissertation.

### 2.2.2 Contention Management

TSX implements Write-Buffering for its shared access tracking and the L1 cache of each physical core is used to hold the corresponding transactional read-sets and write-sets. It is also worth noting that reads and writes are tracked at cache-line granularity rather than individual words. By designing the HTM system to match existing chip-set specifications, TSX is able to leverage existing cache architecture and cache coherence protocols for HTM operation [32].

Although details are scarce, based on analysis and Intel’s historical product specifications, MESIF or a similar variant is the cache coherence protocol used in Haswell [19]. MESIF in conjunction with a write-back cache system means that any cache invalidation will be immediately received by other threads.

Based on available documentation [32], we identify TSX’s CD policy to be pessimistic (eager) and CR policy to be abort-readers. By deploying such policies, TSX can maintain strong atomicity guarantee. If an address residing in a transaction’s read-set or write-set is modified by another thread (does not have to be inside a transaction), then it will cause an abort of the current transaction.

### 2.2.3 Progress Guarantee and Software Fall-back

Speculative execution in TSX is simply “speculative” and can be aborted due to variety of reasons. If and when a transaction aborts, it is the developer’s responsibility of either
roll-back, and retry a transaction or give up on speculative execution and fall back to a purely software execution path. Currently, the simplest method of falling back is to have a single global fall-back lock and perform coarse-grained lock-based execution [22].

However, critical sections executed in software through locks are not protected by strong atomicity. Another running hardware transaction may commit changes to shared memory without being ”detected” by the thread executing in pure software. Therefore it is crucial to perform a lock check after a TSX transaction has started. Doing this lock check is to protect the case where a hardware transaction executes while a software transaction (or locked critical section) is executing. Essentially the strong isolation provides protection to the hardware transactions but a software transaction (or critical section) does not enjoy the same protection from hardware transactions.

As a result, application or library developers must guarantee the system at any given time execute only 1 software transactions (critical section) and no hardware transactions or multiple hardware transactions and no software transactions at all but never both. We describe in detail how the fall-back path is implemented in Section 3.3.

2.2.4 TSX Abort Causes

TSX transactions utilizing both HLE and RTM can be aborted due to a variety of reasons [9]. In addition to real shared data conflicts, TSX enabled transactions can easily be aborted due to various other reasons, they are listed below:

1. Real Data conflicts as described in 2.2.2

2. Cache resource overflow: when the size of a transaction’s read-set, write-set, or a combination of either along with other data residing in the L1 cache becomes too big and causes a cache eviction.

3. False sharing: as described in 2.2.2 TSX tracks shared data at cache-line granularity.

4. Conflicting accesses to the global fall-back lock: as described in 2.2.3

5. Unsupported instructions: such as OS services, I/O, exceptions or certain system calls that might have irreversible effects TSX cannot fully roll-back.

These abort causes can be identified with reading the status codes in the EAX register [9] and they are used throughout our library designs for code path changes.
Chapter 3

TSX-Based Library Design

As discussed in Section 2.1, TM systems are complex in nature and can be comprised of many individually designed components. In order to ensure consistency, our library implementations make use of a set of shared TM system components. Specifically, throughout our study, we use the same: experimental hardware platform, application programming interface (APIs), and progress guarantee implementation. The rest of this section describes each component in detail.

3.1 Hardware TM Platform: Intel Haswell

To focus our efforts and limit the scope of this dissertation, the hardware environment for our study is Intel’s Haswell platform with TSX support. We perform all our experiments on a desktop PC with 4 2-way Hyper-threaded Intel i7-4770 cores at 3.4GHz and 12GB of RAM, 4x32KB L1 caches, 4x256KB L2 caches and a shared 8MB L3 cache. Cache-line sizes are 64 bytes each which in turn enforces TSX RTM tracking granularity at 64 bytes. As mentioned in Section 2.2.1 we limit our experiments to RTM in order to take full advantage of being able to customize the software fall-back path upon transaction aborts.

Compiler and operating systems (OS) consistency for TSX execution on Haswell are ensured by exclusively using GCC 4.8.1 on a 64 bit Ubuntu Linux with 3.5.0-40 kernel. All experiments in this dissertation have threads that are bound to processors ensuring each thread run exclusively on its (Hyper-threaded) core. We note that having only 4 physical cores on our hardware platform limits the extent of our scalability studies. There is speculation regarding additional chip-sets and platforms Intel may release in the future supporting more than 4 physical cores [40]; however they are beyond the scope of this dissertation.
3.2 Application Programming Interface

In addition to the hardware execution platform, we also leverage a set of easy to use APIs from our in-house libTM STM library. libTM [24, 25] is a highly-customizable STM library written in C++ with an intuitive API, and a flexible core that enables research of a variety of TM related studies.

The libTM API allows fast and flexible parallelization of applications through a set of macro expansions and function calls. The ready-made functions provide convenient features such as: thread management, implicit barriers, and transactional scope setup. We describe the generic API calls used throughout our experiments below.

First, thread creation and management can easily be done; thread pool management and implicit barrier creation can be performed through the following macros:

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

The PARALLEL_EXECUTE macro performs parallel execution of parallel_func with a predefined thread count; it takes the parallel_func function pointer to any function in the application, the id of the thread executing it, and any other arg as arguments. parallel_func may contain multiple transactions, or call other functions that may contain transactions.

For any critical sections executed as transactions, transactional scope can be set up using:

- BEGIN_TRANSACTION();
- END_TRANSACTION();

By using these two macro expansions, the API is able to keep track of the where and under what circumstances a transaction has begun. Such information are used extensively in our specific Hybrid library implementation in order to perform the necessary roll-back actions on aborts.

In addition to the two set of APIs described, there is also another set of interfaces for tracking shared variable reads and writes. As explained in Section 2.2.2, TSX performs variable tracking automatically through hardware, thus no additional software tracking is necessary. However, the APIs used for access tracking are used extensively for our Hybrid TM library implementation and will be explained as part of the Hybrid TM library design in Chapter 5.
3.3 Software Fall-back Path Implementation

As described in Section 2.2.3, RTM instructions alone are not enough to properly execute application code in hardware transactions. There must be additional code at either library or application level to provide a software fall-back path in order to guarantee overall application progress. The design of the software fall-back path is a topic of research for many; we have chosen the simplest form as recommended by Intel: with a global fall-back lock [9]. Having a simple fall-back path allows us to focus our investigation and isolate the effects of TSX itself. Our Hybrid TM implementation includes a much more complex design of the software fall-back path.

In order to conform to the API described in Section 3.2, we integrate RTM-specific instructions into the transaction related macro expansions. This allows us to utilize existing benchmarks and applications without making significant code changes. Listing 3.1 and Listing 3.2 show the pseudo-code transformations for our implementation of the
Listing 3.1: Exposed API to application developers.

```
BEGIN_TRANSACTION();

// Execute critical section code

END_TRANSACTION();
```

Listing 3.2: RTM hardware transaction implementation with single global fall-back lock.

```
HW_Transaction:
    XBEGIN {
        if (isLocked(fallback_lock)) {
            XABORT;
        }
        // Execute critical section code
        XEND
    }

Abort_handler:
    // branch here upon XABORT
    abort_counter++;

    if (abort_counter < MAX_ABORTS_ALLOWED) {
        goto HW_Transaction;
    } else {
        lock(fallback_lock);
        // Execute critical section code
        unlock(fallback_lock);
    }
```

transactional API.

To further assist in understanding the code path a transaction might take, Figure 3.1 shows a flow diagram of the possible code paths. Our TSX API interface implementation allows us to attempt a hardware transaction multiple times before falling back on to the default fall-back path and acquiring the global fall-back lock. As explained in Section 2.2.3, threads executing the critical section in software mode are protected from other software transactions but not from other hardware transactions; A hardware transaction must proactively self-abort if it determines that another thread holds the global fall-back lock (as shown in Listing 3.2 line 6).
Chapter 4

TSX Modelling and Characterization

With the TSX based library in place, we model and profile Intel’s Transactional Synchronization Extensions (TSX) by conducting two case studies. Our goal is to understand the behaviour, advantages, and limitations of TSX in order to derive a set of application optimization guidelines and suggestions. The first case study will focus on extracting TSX performance trends and the second will focus on how these extracted trends can be applied to a more complex application through manual tuning and optimization.

4.1 Case Study 1: Array Access Micro-benchmark

In this case study, we provide detailed descriptions of how we characterize TSX performance using a simple array access micro-benchmark and discuss the trends we observe based on the collected results. The models will also provide a list of limitations of TSX that developers should be made aware of when optimizing application code.

As shown in Listing 4.1, the micro-benchmark maintains a one dimensional integer array where its elements are accessed through either reads or writes based on pre-generated access patterns. The micro-benchmark accepts three 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:

1. **Random Mode**: Each thread accesses txn_size elements in the main array based on a pre-generated array of randomly chosen indexes.

```c
ArrayAccessBenchmark() {
    GenerateAccessPattern();

    for every iteration in N iterations {
        BEGIN_TRANSACTION();
        // reads and writes based on access pattern
        AccessArray();

        END_TRANSACTION();
    }
}
```

2. Contiguous Mode: Each thread accesses \textit{txn\_size} elements in the main array in a contiguous fashion starting from a pre-generated, randomly chosen starting index.

In both modes, read and write accesses are randomly assigned based on \textit{write\_ratio}.

### 4.1.1 TSX Modelling Results

![Abort rate vs. txn size vs. write ratio for the array access micro-benchmark using 4 threads, 0 transaction retries, and random access pattern.](image)

**Figure 4.1:** Abort rate vs. txn size vs. write ratio for the array access micro-benchmark using 4 threads, 0 transaction retries, and random access pattern.
For our profiling study, we set the main integer array to 3000 elements and the problem size to 5 million iterations across 4 threads mapped to individual cores. Through varying the tuning parameters, we aim to identify possible trends and effects of each parameter on TM-related metrics. Specifically we use the following as our performance metrics: 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.

Next we present a representative subset of our characterization results.

Figure 4.1 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 the 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 as contiguous accesses take better advantage of cache locality.

Figure 4.3 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 5.7 million shared accesses per second. Our findings show that overall performance does not necessarily correlate with the smallest transaction size. Further experiments performed on the array access micro-benchmark using a single thread re-
Figure 4.3: Abort rate vs. txn size vs. write ratio for the array access micro-benchmark using 4 threads, 0 transaction retries, and random access pattern.

Figure 4.4: Throughput vs. txn size vs. write ratio for the array access micro-benchmark using 4 threads, 0 transaction retries, and contiguous access pattern.

veal that the overhead of entering and exiting a RTM transaction can be as much as 600 milliseconds on our hardware platform. The RTM overhead is amortized in larger
transactions thus revealing a peak in overall transaction throughput.

Figure 4.5: Abort rate vs. write ratio and the number of retries for the array access micro-benchmark with contiguous accesses, 4 threads.

Finally, Figure 4.5 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.

### 4.1.2 TSX Performance Trends

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

1. The size of the read and write-sets directly correlates with abort rates and smaller transactions exhibit higher likelihood of committing. Unfortunately, the exact size limit is unknown and difficult to calculate due to the complexity of the cache coherence protocol. However, through our experiments, we estimate that with the current TSX resource limitations, transactions would need manual or automatic optimizations for appropriate sizing for most non-trivial applications.

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. Extremely small transaction sizes should be avoided as the overhead of the RTM instructions can impact overall performance significantly.
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.

These summarized trends provide a general guidance direction for our manual application optimization and Hybrid TM library implementation.

### 4.2 Case Study 2: Moldyn, A Molecular Dynamics Simulation Benchmark

Listing 4.2: Simplified Pseudo code for Moldyn.

```plaintext
Moldyn() {
    InitializeMolecules();

    for every time step in N timesteps {
        Once every Kth time step {
            UpdateNeighbours();
        }

        ComputForces();

        Update();
    }
}
```

In the second case study, we further characterize TSX by translating and verifying the summarized trends from Section 4.1 into manual optimization techniques. We evaluate the performance impact of TSX using a more complex application: Moldyn. A detailed description of different versions of Moldyn and our manual code transformations will be presented. Specifically, we will present evaluation results comparing different versions of Moldyn in terms of run-time, scalability, abort rate and list additional details about TSX.

Moldyn is a molecular dynamics simulation in three dimensional space that iterates over a number of time steps. Starting from an initial molecule distribution, it computes
Listing 4.3: Pseudo code for unoptimized sequential version of ComputeForces in Moldyn.

```plaintext
ComputeForces() {
    For interacting molecule pair i, j {
        // Compute 3D displacement
        // Calc. & Update 3D force vectors for i
        // Calc. & Update 3D force vectors for j
        // Accumulate energy totals
    }
}
```

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. Listing 4.2 shows a simplified sequential version of how Moldyn works.

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.

### 4.2.1 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 two different synchronization mechanisms: Pthread mutex locks, and HTM. We list the corresponding parallelized baselines accordingly below:

- Pthread mutex locks and explicit barriers are used to create coarse-grained lock based `lock.coarse` and fine-grained lock based `lock.fine`.

- There are also 2 HTM baselines using the TSX-based library as described in Chapter 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(\texttt{tsx.fine}) accessing only 1 molecule is evaluated.

### 4.2.2 Optimizing Moldyn

Listing 4.4: Pseudo code for the optimized lock based version of \textit{ComputeForces} in Moldyn.

```plaintext
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)
}
```

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 \texttt{lock.opt} and \texttt{tsx.opt}.

Through profiling and parallelizing Moldyn, we identify several critical sections within the code base. Using the function \textit{ComputeForces} as an example, we list 4 categories of code transformations and optimizations below. Optimizations 1 and 2 apply to both \texttt{lock.opt} and \texttt{tsx.opt}, whereas optimizations 3 and 4 apply more directly to \texttt{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, such as, total energy 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 pseudo-code for the unoptimized sequential starting point in Listing 4.3, `lock.opt` in Listing 4.4 and `tsx.opt` in Listing 4.5.

---

Listing 4.5: Pseudo code for the optimized TSX version of `ComputeForces` in Moldyn.

```c
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

        // optimal transaction size
        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();
}
```
4.2.3 Moldyn Results

Next, we compare the results of optimized Moldyn against baseline versions. Figure 4.6 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 4.7 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 4.1 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 4.8 and 4.9 show the effects of increasing RTM transaction retry count using tsx.opt as an example. 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 of the total transactions from approximately 90% to approximately 10%.
Figure 4.7: Moldyn speedup comparison when using different number of threads. The Y axis shows normalized speedup over the sequential version as a baseline.

<table>
<thead>
<tr>
<th>Version</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>txns</td>
<td>abort rate</td>
<td>txns</td>
<td>abort rate</td>
</tr>
<tr>
<td>tsx.coarse</td>
<td>62</td>
<td>52%</td>
<td>124</td>
<td>52%</td>
</tr>
<tr>
<td>tsx.fine</td>
<td>211M</td>
<td>0%</td>
<td>211M</td>
<td>11%</td>
</tr>
<tr>
<td>tsx.opt</td>
<td>75M</td>
<td>0%</td>
<td>75M</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 4.1: Transaction count and abort rates for Moldyn using Intel TSX/RTM.

Figure 4.8: Speedup vs number of retries for Moldyn. The Y axis shows normalized speedup over the version with no retries as a baseline.
Further analysing the presented results, we show that with both \texttt{lock.coarse} and \texttt{tsx.coarse}, there is very little performance improvement. \texttt{tsx.coarse} tracks the performance of \texttt{lock.coarse} simply due to all transactions aborting and taking the fallback path turning the \texttt{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 [4.6]). While the \texttt{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 \texttt{tsx.fine} version does not perform well and results in slowdowns in all cases. We further discuss the reasoning in Section 4.2.4.

### 4.2.4 Analysis of Optimizations

In this section, we analyze in detail the obtain Moldyn results and provide reasoning behind our applied optimizations.

#### 4.2.4.1 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 displace-
ments 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, data contention is also preva-
alent in Moldyn due to conflicting memory accesses. 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 part-
ners 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 4.6 performance improvements between lock.fine and lock.opt
versions are mainly due to privatization. Privatization also plays a major role in produc-
ing a meaningful speedup in the case of the TSX based version. This leads us to believe
that applying privatization appropriately can produce better performance in most appli-
cations.

4.2.4.2 Optimal Transaction Size

Section 4.1.1 shows that the 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 4.1). It is
apparent that the large size of these transactions is stressing the L1 cache resources,
leading to repeated aborts.

At the opposite end of the spectrum is tsx.fine, where the abort rate decreases if
we map the fine-grained locks to small transactional regions containing 1 iteration of
only 1 molecule update; on the other hand, 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 as shown in Table 4.1.

These results are consistent with the ones obtained with the array access micro-
benchmark shown in Figure 4.3 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 transac-
tion 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 4.2.3.

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.

4.2.4.3 RTM Transaction Retries

Similar to our findings for the array access micro-benchmark, tuning the number of retries for the RTM hardware transactions can provide significant performance improvements for Moldyn. As shown in Figures 4.8 and 4.9, 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. [42], one might obtain better results by further tuning.

4.2.4.4 Programmability and Programming Effort

Based on the modelling results and guidelines identified in this chapter, we observe that applications optimized with TSX can provide comparable performance to lock based counterparts. However, despite the success in creating performance-optimal code, the manual optimization process is rather cumbersome. Changing critical section to reduce transaction size and aborts require deep understanding of application code and data structures. For an application like Moldyn with less than 1000 lines of code, it took weeks of code modifications and re-writes to achieve the performance and correctness targets. In order to allow and promote adoption of TM for larger projects and code bases, a need clearly exists for a system with added programmability.

4.3 TSX Limitations and Behavior

Based on our modelling results in 4.1.1 and summarized trends shown in 4.1.2, we discuss in detail the various intricacies of TSX behaviour and suggest possible optimizations
developers can make use of to fully exploit speculative execution.

### 4.3.1 Cache Size Limitations

As noted in the previous section, due to L1 cache size limitations, the sizes of read-sets and write-sets belonging to each transaction/thread is limited. While there is no definitive way to determine the exact size of the set sizes due to the sheer complexity of cache hierarchy of Intel processors. We can combine our characterization results with factors such as cache line size, and set associativity (8 in the case of Haswell L1) and estimate that set sizes generally cannot exceed 5-10 cache-line sizes. This places constraints on applications with large critical sections and will almost always abort due to resource overflow when executing in TSX mode.

Side effects from other operations: invoking dynamic library function for the first time (software system needs to invoke dynamic linker to resolve symbols). This may lead to excess amounts of data accesses the 1st time it happens.

### 4.3.2 Lemming Effect

As described in Section 3.3, we use a global software fall-back lock to provide application progress guarantee. When the fall-back lock is not free it propagates aborts throughout the entire system; this is commonly referred to as the Lemming Effect \[23\]. In order to prevent such adverse negative impact on our application execution, we changed our system to retry for hardware transactional execution if and only if the software lock has become available. This way we avoid wasting unnecessary CPU cycles on transactions that will surely fail and we avoid thrashing the cache of threads that are already executing in software-locked fall-back mode.

### 4.3.3 Statistics Tracking with TSX

In order to fully characterize TSX behavior, it is very important for us to collect statistics on the TSX execution. However, we note a very important and worthy issue with using counters inside transactions. Shared counters are essentially another shared variable introduced inside transactions and can cause high abort rate. It is advisable for application developers to use thread local counters for statistics collection although using large number of statistics variables will in addition impact TSX transaction sizes.

As a compromise, it is advisable to use the EAX register which stores several categories of TSX execution information and can be used to further analyze TSX behavior.
These register values should be read outside of regular transactional execution to ensure minimum contamination of hardware transactions.
Chapter 5

HyTM: Hybrid TM Library Design

In this section, we provide the design and implementation details of our Hybrid TM library: HyTM. The goal in creating HyTM is to automate the manual optimization techniques identified with improved programmability and added support for larger, more complex parallel applications. Through using additional layers of fall-back paths, HyTM transparently guides transactions towards best-effort speculative execution with reduced transaction sizes.

The design of HyTM is an extension of the TSX-based library as described in Chapter 3. The APIs are mostly consistent from before but with several additional requirements. The RTM and the global software fall-back layers are also re-used with additional hybrid fall-back layers inserted.

The rest of this section will describe in detail HyTM’s access tracking, and conflict detection mechanisms, fall-back layer designs, and correctness guarantee techniques.

5.1 API Additions

HyTM’s API leverages the same set of macros as described in Section 3.2 with several updates and additions. Thread and thread pool management are done in the exact same way; transaction scope macros \texttt{BEGIN\_TRANSACTION} and \texttt{END\_TRANSACTION} retain the same syntax, but with revamped implementations. There are newly introduced interfaces, which allow HyTM to perform accessing tracking, conflict detection and atomicity guarantee in software without significant effort from the application developer. In this section we describe the design of the added interfaces and their usage instructions, and in later sections we describe how they are used in HyTM operation.
5.1.1 Access Tracking with TM Wrapped Data Types

The first addition to the API is the mandatory requirement of converting all shared data objects to \textit{tm\_types}. \textit{tm\_types} are wrapper objects that allows HyTM to track shared accesses, collect statistics, and maintain a pointer to a version tag for atomicity guarantee. Figure 5.1 provides a graphical representation of \textit{tm\_types} design and Listing 5.1 provides sample code for how \textit{tm\_types} is implemented. By converting objects to \textit{tm\_types}, HyTM can intercept the read/write calls in software by overloading common operators and performing the corresponding transactional operations within the following actions:

1. Tracking of all read and write accesses to shared variables when necessary in order to create corresponding read and write sets. Details of the access tracking operations in each fall-back layer are described in Section 5.2.

2. Update meta-data such as statistics counters for each shared data variable for debugging, and code analysis.

3. Maintain a pointer to a version tag that acts similar to a lock and is used to ensure program correctness. Version tags are presented in detail in Section 5.1.2.

For simplicity, the implementation of our \textit{tm\_types} wrapper interface is based on the previously developed libTM library [24, 25]. A set of primitive types such as \textit{tm\_int}, \textit{tm\_double}, and several others are provided for programming convenience. Custom data structures and objects application can also be easily made (\textit{tm\_types}) using generics.
Listing 5.1: \textit{tm\_types} wrapper interface.

\begin{verbatim}
template< typename T >
class tm_type {
private:
    volatile T original_object;
    ver_t * version_tag_ptr;
    stat_t meta_statistics;

public:
    tm_type() {
        init_ver_tag(version_tag_ptr);
        init_stat_counters(meta_statistics);
        // also initialize original\_object of type T
    }

    /**
     * Overloaded assignment operator with object of type T
     * called inside atomic region to ensure correctness
     */
    T& operator=(const T &new_value) {
        if (access_trap_flag) {
            // record new value in write_set;
            return write_to_write_set(&original_object, new_value);
        } else {
            // increment version tag and assign new value.
            ++(version_tag_ptr -> version_tag);
            return (original_object = new_value);
        }
    }
}
\end{verbatim}

5.1.2 Conflict Detection with Version Tags

As described in Section 5.1.1, each shared data variable converted to \textit{tm\_types} contains a version tag pointer that points to a version tag object. Each shared data variable, by default, points to its own version tag or can be manually altered to point to a common version tag shared with others. Figure 5.2 depicts a graphical representation of how version tags can be shared. Manually assigning version tags is optional, and can be done using the \textit{assign\_tag()} function.

In HyTM, we use the version tags associated with each shared variable as a unified way of performing conflict detection, resolution and overall correctness guarantee. The version tags belonging to entries in the software read-set are collected to form a reduced a software-based read-set. Every time a shared data variable is written to, its version tag
Figure 5.2: Multiple *tm_type* wrapped application objects sharing the same version tag, which acts similar to a lock.

is atomically incremented either inside a RTM transaction or in a locked critical section. When a transaction running on a different thread accesses the same shared variable, the current version number in the version tag is read and collected into the software-based reduced read-set. Any changes to the version number in the recorded version tag will invalidate any read-sets it belongs to and result in an aborted transaction.

In addition to using version tags as individual object locks, coarsening version tag granularity by sharing them amongst multiple *tm_type* is a key to reducing transaction and read-set sizes. The specific details of how transaction sizes can be reduced in certain HyTM layers are described in Section 5.2. In this dissertation, our experiments utilize manual version tag coarsening. However, application assisted and automatic version tag coarsening is part of our research plan moving forward.
5.2 Multiple-layer Fall-back Mechanism

HyTM leverages a large portion of features from the TSX-based library described in Chapter 3. In addition to the common APIs, HyTM extends the two existing transaction layers as described in Section 3.3: the RTM hardware only layer, and the last resort fall-back of locking the software global lock for the entire critical section execution. HyTM also inserts two additional fall-back layers between the existing two layers in order to provide automated transaction and read-set size reductions. A side benefit of having the additional layers is that they help reduce the impact of the Lemming effect by eliminating or delaying the need to acquire the global fall-back lock.

In total, HyTM contains four transaction layers: hardware only, hybrid with hardware commit, hybrid with software commit, and software only. As the names imply, the two hardware-based layers share the common trait where commits are done with RTM transactions; the two software-based layers use locks to execute critical sections in order to ensure atomicity. An updated overview of the transaction flow path based on Figure 3.1 is shown in Figure 5.3.

In the rest of this section, we provide a detailed description of each HyTM transaction layer’s implementation, abort criteria, expected performance, as well as progress and atomicity guarantee design.

5.2.1 Layer 1: Hardware Only

The first layer of HyTM’s transaction execution path uses RTM for hardware only speculative execution. As described in Section 2.2.1, shared accesses do not need to be explicitly tracked in software as TSX makes use of existing L1 cache space for maintaining read-sets and write-sets. As a result, only XBEGIN and XEND instructions are needed to denote the begin and the end points of the hardware only transaction. The abort handler for HyTM’s hardware only layer builds on top of the existing TSX-based library as well. In addition to the RTM transaction maximum retry limit, we also make use of the transaction status codes read from the EAX register as described in Section 2.2.4. Based on trends derived from our modelling results and the status code in the EAX register, a RTM transaction can take one of several execution paths as shown in Figure 5.3 and in Listing 5.2:

- **Successfully commits with XEND**: means that the critical section has completed execution speculatively and all writes appear atomically to all threads.

- **Global software fall-back unavailable**: indicates that another transaction has
Figure 5.3: Hybrid TM transaction flow diagram.
taken the software fall-back lock and is currently executing the critical section in one of the two software modes. In such case, the current RTM transaction is aborted with the `XABORT` instruction and attempts to retry if the maximum number of allowed retries has not been reached. HyTM will wait until the lock becomes available before attempting to retry the speculative execution again; doing so avoids triggering the Lemming effect, as described in Section 4.3.2, and prevents wasting HTM retry attempts.

- **Aborted due to conflicts**: a conflicting access caused by accessing the same cache-line in memory has occurred. The RTM transaction will also attempt to retry if possible.

- **Maximum retry limit reached**: if the RTM transaction has exhausted all its allowed retry attempts, it aborts from the hardware-only layer and falls back to layer 2.

- **Aborted due to resource overflow**: if the RTM transaction size is too long, its read-sets or write-sets might overflow the available L1 cache space. As we have shown in Chapter 4, retrying such long transactions is a futile task. HyTM immediately aborts to layer 2 avoiding attempting to retry the current RTM transaction.

- **Incompatible instructions or other abort causes**: As described in Section 2.2.4, incompatible instructions or system calls such as `malloc()` will almost certainly abort a RTM transaction prematurely. The same transaction will waste retry attempts as it will abort over and over at the same point in code. In such cases, HyTM eliminates retry attempts after the first abort and falls back to layer 2. Executing incompatible instructions is a common problem in TM systems in general.

The overhead of the hardware only layer is minimal due to not having the need to explicitly track shared data accesses. However, it is worth noting that all writes to `tm_types` wrapped objects increment their version tags which may increase RTM transaction size and execution time. The increased transaction size is a trade-off HyTM makes in order to ensure overall application correctness.

### 5.2.2 Layer 2: Hybrid with Hardware Commit

Aborted transactions from the previous hardware only layer will fall-back to layer 2: hybrid with hardware commit. As the name implies, this layer is a hybrid of software and
Chapter 5. HyTM: Hybrid TM Library Design

5.2.2.1 Compute Phase with Automatic Privatization

In the first phase of layer 2 as shown in Figure 5.4, the entire transaction is executed without using RTM or acquiring any locks. All shared reads and writes are trapped by hardware execution of the critical section. Utilizing the shared version tag mechanism described in Section 5.1.2, HyTM creates smaller, software-based read-set from the aborted hardware transaction in layer 1 and a software-based write-set. The newly created read and write-sets form a reduced-sized transaction, which is attempted in hardware. There are two main goals for using the hybrid layer with hardware commit: increase the likelihood of hardware transaction committing and reduce the impact of the Lemming effect, by not acquiring the global fall-back lock.

Based on the modelling results shown in Chapter 4, unoptimized and long running RTM transactions are more likely to abort due to resource overflow. Intuitively, the write-set inside a critical section cannot be eliminated and any improvements must come from reducing the size of the read-set. HyTM performs such read-set size reduction by tracking version tags associated with reads rather than the actual values. As depicted in Figure 5.4, layer 2 starts by running a compute phase to execute the critical sections without committing any changes to shared memory. The compute phase collects writes into a software-based write buffer and tracks version tags associated with read shared variables into a read-set. Next, a commit phase is attempted in hardware to re-validate all entries in the read-set and "flush" all entries in the write-set in one atomic operation.
Chapter 5. HyTM: Hybrid TM Library Design

the overloaded operators of the \texttt{tm} types and processed by HyTM. The access tracking strategy we implemented is Write-Buffering (WB) with optimistic conflict detection (CD) as described in Section 2.1.2 and 2.1.3. Read and write accesses to the shared memory locations are documented in detail below:

1. **Read Accesses**: all the shared reads are tracked but they are no longer stored in a read-set, only the associated version tags of those reads are stored in a software-based read-set. The size of the read-set can be greatly reduced if multiple shared memory variable share the same version tag (as depicted in Figure 5.4).

2. **Write Accesses**: as part of the WB conflict detection design, shared writes are not visible to other threads during the compute phase. All shared writes are recorded in a software-based write-set and each write-set entry records the memory address, and the updated value of the variable that will become visible should the transaction successfully commit. HyTM avoids duplicate write-set entries by searching and replacing entries on subsequent writes within the same transaction.

With the automatically privatization techniques described above, HyTM avoids having to acquire the global software fall-back lock and thus allowing multiple threads to perform the computation completely in parallel. Once a new set of read-set and write-set is constructed, HyTM process to the commit phase of layer 2.

### 5.2.2.2 Commit Phase with Reduced-size RTM Transaction

Upon completing the compute phase, HyTM holds a read-set of all the version tags associated with the shared reads and a write-set of all shared writes. HyTM’s next task is to make the write-set visible to all other threads atomically; this is done with a shorter RTM transaction as depicted in Figure 5.4.

In order to fully guarantee atomicity, the commit phase must re-validate all entries in its read-set and "flush" all entries in its write-set in one atomic operation. Thus two sets of operations are done once \texttt{XBEGIN} has been called to start the RTM commit transaction:

1. **Read-set Validation**: involves comparing the current version tag values of the shared reads against the recorded version tag values. A difference would indicate that another thread has made changes to the read value which results in an abort.

2. **Write-set Flushing**: if all entries in the read-set have been validated, HyTM proceeds to "flush" the entries in the write-set into their actual locations in memory.
RTM performs the action by copying the write-set entries into the L1 cache and in turn committing them to the main memory if the transaction succeeds.

As Figure 5.4 depicts, the size of the RTM transaction in the commit phase of layer 2 has been reduced, primarily because of shared version tags. We expect applications to be able to take advantage of this feature and resolve the RTM resource overflow limitation in order to successfully commit in hardware.

With any RTM transaction, the commit phase using hardware faces the same pitfalls as the hardware only layer: progress is not guaranteed. Aborts can occur due to the same reasons as described in Section 5.2.1. The abort handler of the hybrid with hardware commit layer is shown in Listing 5.3 and has an identical design to the hardware only layer with one exception: read-validation aborts. Aborts caused by read-validation failures indicate that the compute phase of layer 2 needs to be re-run. Any retries by HyTM will need to start from the beginning of layer 2. Currently, HyTM aborts directly to layer 4 when encountering read validation failures and retrying the entire hybrid transaction is a topic of future research. As long as read-set validation does not fail during the RTM commit phase, the transaction can be retried in hardware, until the maximum allowed number of retries has been reached. Aborts due to reasons other than read-validation failures are handled by HyTM by falling back onto layer 3: hybrid with software commit layer, which is described next.

5.2.3 Layer 3: Hybrid with Software Commit

The third fall-back layer in the HyTM design involves the same compute phase but with the commit phase done in software. However, currently, HyTM does not allow retry of the two hybrid layers; therefore, layer 3 is only used as a software fall-back for the commit phase of layer 2 as shown in Figure 5.4 and described in Section 5.2.2.2.

Transactions fall back to layer 3 due to aborts from the RTM commit phase of layer 2. Although the entries in the write-set have not been successfully committed, the read-set of all the version tags is still intact. In such scenarios, HyTM simply attempts to lock the global fall-back mutex lock and re-execute the commit phase as one single critical section.

It is worth noting that the read-set constructed in layer 2 still needs to be verified in order to prevent any version tag updates between the commit phase aborting in layer 2 and the commit phase acquiring the lock in layer 3. If read-validation fails, HyTM aborts directly to layer 4 for last resort execution.
5.2.4 Layer 4: Software Only with Single Global Fall-Back Lock

Once HyTM has exhausted all possible speculative and hybrid execution opportunities without successfully committing, it falls back to the last layer: a last resort pure software layer using the single global fall-back lock. The entire transaction will execute in one single coarse critical section thus preventing all other threads from making progress until the current transaction is completed.

The design of layer 4 in HyTM is identical to the software fall-back path design described in Section 3.3. Falling back all the way to the software global lock is obviously undesirable and can result in serialization of the application’s critical section. But as before, the global lock based fall-back path is a necessity that HyTM must include in order to guarantee overall application progress.

5.2.5 Implementation Details

In order for all HyTM layers to perform as designed correctly and efficiently, several implementation decisions are made.

The read-sets and the write-sets are implemented using linked hashtables with bloom-filters for fast searching and iteration. Both sets are allocated on a per thread basis so they can be reused for multiple transactions throughout their lifetime.

As described in Section 5.2.2.2, entries in a transaction’s read-set can cause transaction aborts if any of them become invalidated due to writes from another transaction. Typically the validation process is done in the commit phase of the two hybrid layers. Optionally, during a read access to a shared memory location, HyTM can immediately check whether the version tag associated with the read already exists in the read-set. If a match is found and the version number is found to have been incremented, the current transaction is aborted and can either be retried or fall-back directly to layer 4. Using such pessimistic conflict resolution techniques may provide run-time savings but can also incur additional overhead under certain scenarios.
Listing 5.2: HyTM layer 1: hardware only layer using RTM

```c
HW_Transaction:
   XBEGIN {
      if (isLocked(fallback_lock)) {
         XABORT;
      }
      // Critical Section
      XEND
   }

Abort_handler:
   // branch here upon XABORT
   abort_counter++;
   if (abort_counter < MAX_ABORTS_ALLOWED) {
      // Retry RTM transaction
      goto HW_Transaction;
   }
   abort_status = EAX_REGISTER;
   // Abort due to conflicts
   if (abort_status == CONFLICT) {
      // Retry RTM transaction
      goto HW_Transaction;
   }
   // Other abort causes
   if (abort_status == RESOURCE_OVERFLOW) {
      abort_to_layer_2();
   }
   if (abort_status == OTHER) {
      abort_to_layer_2();
   }
```

Listing 5.3: HyTM layer 2: hybrid with hardware commit using RTM

```
Compute_Phasen:  
    // Execute critical section
    construct_read−set(); // with read version tags
    buffer_writes(); // into write−set

Commit_Phasen:
    XBEGIN {
        if (isLocked(fallback_lock)) {
            XABORT;
        }
        // Read−set validation
        validation_flag = validate_read_set();
        if (!validation_flag) {
            abort_to_layer_4();
        }
        // Write−set flushing
        Flush_write_set();
        XEND
    }

    // branch here upon XABORT
    abort_counter++;

    if (abort_counter < MAX_ABORTS_ALLOWED) {
        // Retry RTM transaction
        goto HW_Transaction;
    }

    abort_status = EAX_REGISTER;

    // Abort due to conflicts
    if (abort_status == CONFLICT) {
        // Retry RTM transaction
        goto HW_Transaction;
    }

    // Abort due to other reasons
    if (abort_status == RESOURCE_OVERFLOW) {
        abort_to_layer_3();
    }

    if (abort_status == OTHER) {
        abort_to_layer_3();
    }
```
Chapter 6

Evaluation of HyTM Performance

In this section, we present our benchmark, SynQuake, the experimental setup and evaluation results. Specifically, we describe in detail the parallelization efforts, transactional region composition, and optimizations performed.

Section 6.1 describes the SynQuake benchmark in detail, Section 6.2 provides experimental setup details, application parameter specifics and the evaluation baselines we used. Finally section 6.3 presents the results and analysis from our HyTM evaluation.

6.1 SynQuake Server Parallelization

In order to evaluate our HyTM library, we choose a complex and realistic application as our benchmark: SynQuake Server, an open source port of the popular Massive Multi-player Online Game (MMOG) Quake. Unlike popular TM benchmarks such as STAMP, SynQuake provides more realistic TM usage scenarios and wider range of flexibilities in parameters.

6.1.1 SynQuake Overview

Massive Multiplayer Online Games (MMOGs) have been the subject of study and research for many parallelization paradigms. Their intrinsic complexity with player flocking and hot-spot creations can pose various challenges to programmers. One approach of parallelization of MMOGs is locking, which has become a standard method for protecting shared data.

The concept of locking can be looked at as a type of synchronization mechanism for imposing limits on accessing resources in an environment where there are several threads of execution. There are however complexities that come with the lock-based paradigm
and lead us to question if there are more efficient ways to handling parallel processing. Transactional Memory is one upcoming method in research that attempts to address the complexities of lock-based programming while providing a simple technique for parallel processing.

### 6.1.2 Data Structures

Figure 6.1: SynQuake game map data structure [34].

The SynQuake game world is comprised of different types of entity objects: players, apples, walls, and potentially others. Player objects include many data fields such as: player position, player attributes such as health level, score, player movement, and etc. Entities such as apples are meant to simulate quest scenarios where players might move towards a quest locations in order to obtain power-ups from eating apples. Walls and other stationary entities are meant to introduce additional steps players might need to take in order to reach quest locations. Each game entity object will reside inside a area bounding box that represents their position and their occupied space on the game world map. No more than one active entities such as players can be inside an area bounding box at any given time; and players can only overlap with entities such as apples when an "eating" action is performed.

In addition to performing simple quests such as eating apples, players can also perform other tasks such as moving around, attacking each other and other complex actions. However, in our experiment we eliminate expensive graphical computations such as explosions and game world changes to isolate the effect of parallel execution.

The SynQuake game world is created as a large 2-D game map, where all of the entities
in a game session have a position within the boundaries of the game map. In order to perform efficient searches for all entities that a player interacts with in a given action. The game map is divided into area nodes and represented as an area node tree. The area node tree is a binary space partitioning tree, where each node represents a particular region of the game map (as shown in Figure 6.1b). The tree itself is constructed by recursively splitting the map into sub-units, beginning with the root node, which corresponds to the entire game world. Nodes on lower levels of the tree are then created by splitting the region corresponding to the parent node along a median segment alternately along both the x- and y- axes. This is recursively performed until a predefined tree depth is reached. Depending on an entity’s x and y coordinates on the game map, it may be tracked by any of the leaf or parent area nodes.

6.1.3 Server Application Structure

![Figure 6.2: SynQuake client processing stages](image)

Building on the knowledge of the game’s previous developers [25], the SynQuake server code consists of three stages as shown in Figure 6.2. These three stages are request processing, administrative tasks, and reply processing. The server essentially loops through these three stages forming a server frame or iteration. In the first stage, the server receives requests from clients. These client requests essentially get processed by a worker threads which are responsible for handling their own particular client. The thread assignments are created according to load balancing policies and are updated in the administrative stage. The second stage or administrative stage as seen in Figure 6.2, re-balances load according to the specified load balancing policy. Lastly, the server threads
send updates to the assigned clients in the reply phase of the server frame structure. We can note however that second and third stages don’t need parallelization because these stages only consist of read only operations. Stage 1 will be the focus of our optimization efforts (Section 6.1.6).

6.1.4 Quest Scenarios and Game Configuration

As a TM research benchmark, the SynQuake server application uses a large set of configuration parameters. These parameters allows us to configure the application’s area map size, player count, game duration, player movement speed, player action and movement traces, obstacle placement, player distribution, area node tree lock granularity, and game quest scenarios.

Quest scenarios are especially interesting because they allow developers to simulate large scale game environments with very different contention levels. This not only allows researchers to examine specific low, medium, and high contention game situations; it also simulates player flocking or hot-spots to emulate realistic server applications. Specifically, the application allows developers to simulating one large central quest for all players to attend, smaller localized quests for players nearby, or even moving quests to simulate large scale player group movements. In our experiments we make full use of the flexibility and discuss how these scenarios impact both our library and application behaviour.

6.1.5 SynQuake Parallelization Techniques

In order to easily demonstrate programmability, we leverage a previously parallelized version of the SynQuake game with lock-based synchronization. The code changes we performed were simply converting locked critical sections to transactions using the transactional scope APIs shown in Section 5.1. As we examine the SynQuake code base as described in Section 6.1.3, two transactions are identified: one for adding joined clients and the other for processing client actions. In this dissertation, we focus our efforts on the client action processing transaction as it runs many times throughout an entire game session and is the source of the majority of contention in the application.

The client action processing transaction performs a long list of tasks inside the critical section listed below:

1. Create a potential action area or rectangle for each client player based on his or her requested moving direction, specific action, and speed.
2. Traverse through the area node tree to build a set of all area nodes this player’s action area rectangle overlaps with.

3. Search through every player tracked by every relevant area node and determine whether the current player’s action range rectangle overlap with their current position area rectangle.

4. If the player checks all the players that are close by and determine his movements are clear of any conflicts, he makes the move or commits the move transaction. If any conflicts exist, then the movement action is cancelled.

5. If a player’s movement results in its entity object needing to move a new area node, then the transaction must also perform such actions atomically. In the application source code this involves deleting a player entity object from one area node’s hashtable and adding it to the new area node’s hashtable.

In contrast to the transactional execution described above, the original lock based versions use the identical critical section with two locking options. The first is locking the entire area node tree as a form of coarse-grained locks, when a thread if performing client movement, no other thread can make changes to any node belonging to the area node tree. The second form of locking is to lock only the relevant area node for every client movement operation as a form of fine-grained locking. We use both versions as comparison baselines in our collected results.

### 6.1.6 Manual SynQuake Transaction Optimization

As described previously in Section 6.1.5, the client action processing transaction performs many actions. Similar to Moldyn, in Section 4.2, having such long transactions can easily cause RTM transactions to overflow and abort. Aside from using HyTM for automated transaction size reduction, we also implement versions with manual optimizations for comparison. We note that as with any complex parallel application, optimizing critical sections require considerable care and effort.

The primary candidate functions to be taken out of the transaction critical section are the items 1 and 2 in Section 6.1.5. When processing a client’s action requests, we first create a potential action area outside of any critical section. Next we traverse the area node tree and identify all the area nodes the client’s potential action rectangle overlaps with. Despite the simple nature of such optimizations, a deep understanding of the area node tree data structure is required to guarantee that area node objects will not be re-assigned or move around. In addition, unnecessary aborts can occur due to occasional...
dynamic memory allocation inside transactions for creating data structures tracking area node collections. Additional optimizations are done to reduce the number of malloc() inside the transactions.

6.2 Experimental Setup

In this section, we present our experimental environment as well as benchmark setup details.

As described in Section 3.1, our experiments are run on the same Intel i7-4770 Haswell chip-set with TSX support and are compiled with GCC v4.8.1. All of the results presented are averaged over 3 runs to eliminate the effect of possible statistical errors and run-time anomalies. Other than scalability studies, all of experiments use 4 threads without hyper-threading.

Using the SynQuake benchmark described in Section 6.1, we configure one game experiment with three quest scenarios to simulate different contention levels. The SynQuake game is simulated with 500 synthetically generated clients running 10000 processing cycles. One transaction is attempted in a single processing cycle per each player resulting in a total of 5 million transactions in each game run. The game is configured with a 512 by 512 map and an area node tree depth of 8 (256 tree levels). Each evaluation will be running 3 quest sessions distributed evenly across the total number of server cycles. Players will perform simple short range actions throughout each quest. Each quest session will have a spread of 4 to emulate multiple quest locations. The game setup will allow all players to perform a wide range of actions for extended periods of time as well as simulate various levels of contention when they run into each other. The networking components are disabled in the game in order to eliminate the effects of network delays.

6.2.1 SynQuake Contention Levels and Load Balancing

Quest locations in SynQuake can impact the contention level of the game as described in Section 6.1.4. We use three different scenarios to simulate different contention levels and how they impact TM behaviour as well as overall application performance. Upon game initialization, all player positions are uniformly randomly generated with a uniform distribution. As the game continues, different quest scenarios are introduced:

1. A game with one quest at the center of the map where all players flock towards, shown in Figure 6.3.
2. A game with four quests, one for each game map quadrant. Players flock towards the question location they are closest to, shown in Figure 6.4.

3. A game with no quests where all players move around randomly, shown in Figure 6.5.
In addition to quest scenarios, load balancing algorithms also impacts overall game server performance \[25\]. However, load balancing is beyond the focus of this study and we limit the load balancing algorithm to only spread, which evenly distributes load across all threads \[25\].

6.2.2 Experiment Baselines

Using the above quest scenarios for our experiments, we compare SynQuake results between several different versions:

- The two previously developed lock-based versions of the game as described in Section 6.1.5 are denoted as: \texttt{lock.tree} and \texttt{lock.node}. They will be used as baselines for run-time and scalability studies.

- Three TSX versions with single global fall-back lock are used as transactional baselines: \texttt{tsx.no.opt} represents the lock based versions translated to transactional versions with no optimizations. \texttt{tsx.opt.0} and \texttt{tsx.opt.6} denote heavily hand-optimized and tuned versions with different number of RTM retries as described in Section 6.1.6. \texttt{tsx.opt.6} has a retry limit of up to 6, it is chosen to maximize performance improvements without wasting CPU cycles unnecessarily. Retrying transactions too many times in SynQuake produces diminishing benefits similar to Moldyn, in Section 4.2.3.
• One hybrid TM version called hytm, which is the same unoptimized version of tsx.no.opt, but with the TSX-based library replaced with HyTM.

The hand-tuned TSX versions share the same optimized transaction code, as described in Section 6.1.6. Transaction size is reduced with heavy privatization while carefully maintaining overall application correctness. In addition to hytm, we also include preliminary results of a hand-optimized HyTM version called hytm.opt in Section 6.3.5 to showcase the future potential of HyTM.

6.2.3 Version Tag Assignment

As explained in Section 6.2.2, the HyTM version used in our experiments require very few application-level code changes other than using the HyTM APIs. The main change is manually assigning version tags to shared objects. One may also use the provided statistics collection functions, but this is optional. In our experiments, we limit the version tag granularity to a simple one tag per each game entity object (which may contain multiple shared variables). Finally, all area nodes in SynQuake are associated with their own version tag for simplicity.

6.3 Evaluation Results

Next we present the evaluation results of HyTM using the configured game scenarios described in Section 6.2. The first set of results focus on the transactional versions of SynQuake, where we compare the overall transaction composition, and breakdowns of abort causes. After analyzing the transactional results in detail, we further compare overall application execution time and scalability factors against pre-existing lock based SynQuake implementations.

Specifically, our results analysis is performed with the following goals in mind:

• Quantify the impact of application characteristics and scenarios on the performance of TSX and HyTM.

• Evaluate HyTM’s ability to reduce abort rates and commit a larger fraction of the application transactions in hardware, and its potential to scale with even larger number of cores.

• Determine the performance overhead of the Hybrid TM solution in exchange for minimal programming effort.
Section 6.3.1 compares transaction composition. Section 6.3.2 further studies the reasoning behind aborts at each transaction layers. Section 6.3.4 evaluates the scalability of HyTM when running SynQuake. Finally, Section 6.3.5 show preliminary results of HyTM with manual optimizations as a potential study.

### 6.3.1 Transaction Composition & Distribution

![Graph showing transaction composition](image)

Figure 6.6: SynQuake transaction composition with 1 quest at the center of the game map.

Figures 6.6, 6.7, and 6.8 show the transaction composition of all transactional versions of SynQuake under the three game quest scenarios. The different colored column segments represent where the fraction of overall attempted transactions eventually committed in. As described in section 6.6, all TSX versions of SynQuake will either commit in HW.Only (green) or in SW.Only (red). HyTM introduces two additional layers of Hybrid.w.HW.Commit and Hybrid.w.SW.Commit (blue) as explained in section 5.2 (purple).

As expected with the tsx.no.opt, in all game scenarios, less than 10% of the attempted transactions successfully commit in hardware-only mode, and a vast majority of transactions take the software-only fall-back path. This is primarily due to the long length of the unoptimized transactions. A detailed breakdown of the abort causes are shown in Section 6.3.2.1.
Chapter 6. Evaluation of HyTM Performance

Figure 6.7: SynQuake transaction composition with 4 quests in each of the quadrants on the game map.

Figure 6.8: SynQuake transaction composition where players moving around randomly with no quests.
tsx.opt.0 improves on the results of tsx.no.opt with reduced transaction sizes. With less resource overflow related aborts, the fraction committed in hardware increases for the tsx.opt.0 version of the game by as much as 50%. We also observe that the quest scenario plays a large role in hardware only commit rate of tsx.opt.0. Comparing a game with center quest (Figure 6.6) against a game where there are no quests (Figure 6.8), the percentage of hardware only commits increased from a mere 8% to 52%, a 44% improvement. Similar to tsx.no.opt, we present a more detailed breakdown of abort causes in Section 6.3.2.1.

Further improving on tsx.opt.0 is tsx.opt.6, where we allow the hardware-only transactions to be retried up to 6 times. The limit of 6 retries was chosen to reflect the diminishing return effect described in Section 4.2.4.3. Determining retry limits dynamically is a potential improvement for future studies. As shown in Figures 6.6, 6.7 and 6.8 allowing hardware-only transactions to retry multiple times can drastically improve RTM commit rates. Specifically, in Figure 6.8 an improvement of approximate 90% in commit rate is observed compared to tsx.no.opt. Like tsx.opt.0, tsx.opt.6 version of the SynQuake also correlates with the quest scenario. Comparing the contention level created by 1 center quest against a game with no quests, we observe an increase of the hardware only commit rate from 15% to 94%. These improvements are significant and are also consistent with the Moldyn results in section 4.2.3.

Using the TSX versions as evaluation baselines, we conduct the same experiments using hytm. With the added layers, transactions aborted from the hardware-only mode will not directly go to the software fall-back but rather go through the hybrid layers. With both delayed locking of the software lock, and the additional transaction size reductions in the hybrid layers, as described in Chapter 5, we expect improved hardware only commit rate. hytm shows such immediate improvements even without any application level code optimizations. For the same amount of attempted transactions, hardware-only commits increased from less than 5% to as high as 26% as shown in Figure 6.8. In addition to the improved commit rate of the hardware only layer, the fraction of transactions that are able to commit in hardware in the hybrid layer is also significant. By leveraging the reduced transaction size, and the added benefits of Lemming Effect reduction (as previously introduced in Section 4.3.2), we are able to increase the total percentage of transactions committed using RTM to more than 90% in all game scenarios. Detailed breakdowns of transaction abort causes in each HyTM layer are further presented in Section 6.3.2.

For any transactions that do not eventually commit in hardware using RTM, they take additional fall-back paths to the hybrid with software commit layer and the software
only global fall-back lock layer. In the HyTM version, these 2 cases represent a relatively small percentage of overall attempted transactions.

It is worth noting that these drastically increased hardware commit rates do not necessarily directly translate to improved overall performance as shown in Section 4.1.2 rather these represent a showcase of TSX utilization potential. The main reason for this is the overhead introduced by our write-buffering and read-set tracking. In order to better pinpoint where the transaction commit rate improvements originate, we further dissect the abort causes of each layer in the next section.

6.3.2 Abort Cause Analysis

In this section, we show the collected transaction abort statistics based on the same experiments in order to pinpoint where the commit rate improvements originate.

6.3.2.1 Hardware Only Layer

![Abort cause breakdown of all transactional versions of SynQuake entering the hardware-only layer running a game with 1 quest at the center of the game map.](image)

Figure 6.9: Abort cause breakdown of all transactional versions of SynQuake entering the hardware-only layer running a game with 1 quest at the center of the game map.

All versions of SynQuake utilizing TSX/RTM share a common hardware only layer. This layer contain important information in determining the path all the attempted transactions ended up taking. In this section we analyze in detail the breakdown of the commit rates and abort causes.
Figure 6.10: Abort cause breakdown of all transactional versions of SynQuake entering the hardware-only layer running a game with 4 quests in each of the quadrants on the game map.

Figure 6.11: Abort cause breakdown of all transactional versions of SynQuake entering the hardware-only layer running a game where players moving around randomly with no quests.

Figures 6.9, 6.10, and 6.11 show which paths all the transactions took upon attempting the first hardware only transaction layer. The breakdowns are normalized based on the same number of attempted transactions. The colored columns represent...
the three possible outcomes of an attempted hardware transaction: *Commit*, abort due to *Conflict*, and abort due to *Other* causes. The *Other* category includes resource overflow, fall-back lock unavailable, TSX illegal instructions, and miscellaneous causes which typically indicate TSX transaction will not succeed even if retried [9]. The specific abort reasons match our flow diagram presented in Section 5.2.

As expected, the unoptimized versions *tsx.no.opt* and *hytm* exhibit relatively low commit rates, with *tsx.no.opt* limited to less than 5% in 1 center quest scenario and *hytm* slightly better reaching as high as 26% in the no quest scenario. We observe that the percentage of aborts due to conflicts in *tsx.no.opt* is extremely low at less than 1% consistently. There are two primary reasons: first the unoptimized transactions are long and many transactions cannot reach the commit point before they are forced to abort either due to overflowing L1 cache resources or fall-back lock being acquired by another thread on the software fall-back path; second, all of the aborted transactions of *tsx.no.opt* take up the software fall-back path immediately and attempt to acquire the global fall-back lock. Resorting to acquiring the global fall-back lock effectively serializes transaction execution across all threads thus eliminating the opportunity for memory access conflicts. Although *hytm* executes the same transaction with the same long length, its additional fall-back layers mitigate both mentioned abort reasons of *tsx.no.opt*. By having multiple layers and delays in acquiring the global software fall-back lock, transactions in *hytm* are allowed to make further progress with a higher chance to reaching the commit point. It is evident that many of the transactions do reach the commit point as *hytm* shows as much as 23% more transactions successfully commits in hardware when compared to *tsx.no.opt*.

The optimized versions of the TSX versions of SynQuake unsurprisingly improves in the area of hardware transaction commits, reaching as high as 94% as shown in Figure 6.11.

### 6.3.2.2 Hybrid with Hardware Commit Layer

The second layer of hybrid with hardware commit only applies to the HyTM version of SynQuake as presented in Figure 6.12. The breakdowns are normalized to 100% similar to the hardware only layer, however we note the total number of attempted transactions is different across quest scenarios. Of all the presented plots, we note a very common pattern: the commit rate is very high across all game scenarios. The success comes from several fronts: first, by having a lock-free automatic privatizing computation phase as described in Section 5.2.2.1 the contention for both shared data and the global fall-back lock is much more spread out. Second, the sizes of the read-set and the transaction are
Chapter 6. Evaluation of HyTM Performance

Figure 6.12: Abort cause breakdown of all transactional versions of SynQuake entering the hybrid layer with hardware commit phase running a game with 3 different quest scenarios.

reduced by only having to validate shared version tags of game area nodes. Having the ability to perform such transaction coarsening actions provided great benefits to having transactions successfully commit in hardware.

We also note an addition of the abort category: read buffer validation failures. As explained in Section 5.2.2.2, transactions that fail the read validations during the commit phase are either allowed to retry the entire hybrid transaction, or abort directly to a software only fall-back layer. In this dissertation our experiments do not allow retries of the hybrid transaction for simplicity but it can potentially be a topic of research in the future.

For any other transaction abort causes as described in Section 6.3.2.1, the transactions will fall-back to the hybrid with software commit phase. The automatic privatizing computation phase does not need to be re-executed for these aborted transactions due to the fact that their read buffer entries are still valid until proven otherwise (explained in Section 5.2.3).

6.3.2.3 Hybrid with Software Commit Layer

There will be a relatively small percentage of transactions reaching the third fall-back layer: hybrid with software commit. These transactions failed to commit in hardware due to various TSX abort reasons, but primarily fall-back lock contention as resource
overflow has been mitigated by the reduced transaction sizes. Since their read buffer entries are still intact and valid, they simply attempt to acquire the global fall-back lock and continue execution.

Although the execution of transactions reaching this layer uses a global lock, they can still be aborted as their read buffer entries need to be validated once more in order to ensure overall application correctness. If a transaction aborts in this layer due to read buffer validation failures, it will fall back once more to the last layer of software only execution and re-execute the transaction critical section in its entirety. Figure 6.13 shows the normalized breakdowns of the hybrid with software commit layer. In all of the quest scenarios, we note that at least 65% of transactions successfully commit in this layer. We also observe that the fraction of committed transactions decreases with less contention in the quest scenarios. Even though the observed trend might contrast expectations, it is worth noting that this layer represents only small portion of the overall transaction make up as described in Section 6.3.1. With less than 10% of the transactions reaching the hybrid with software commit stage, the majority of the non-contending transactions have already been committed.

### 6.3.3 Application Execution Time

After analyzing the transaction composition of the different versions in detail, we present the overall application execution time compared against lock based baselines. Fig-
Figure 6.14: Execution time comparison between all transactional versions of SynQuake running a game with 500 players, 10000 cycles, and under 3 different types of game scenarios.

The lock based baseline versions can be categorized into two types as explained in Section 6.1.5: locking the entire area node tree, or locking the relevant area nodes. `lock.tree.no.opt` and `lock.leaves.no.opt` each represent the unoptimized versions of the lock based implementations. As expected, they show slightly longer execution time than their optimized counterparts.

The TSX with single global fall-back lock versions show varying performance results. For high contention levels either due to concentrated game quest scenario, `tsx.no.opt` finished in approximately 37 seconds where `lock.tree.no.opt` and `lock.leaves.no.opt` both finished less than 26 seconds. The reason attributes to the high rate of both fall-back lock contention as well as actual shared data contention slows down overall application performance. For low contention situations such as no quests occurring during the game, TSX based versions can achieve comparable performance, or in the case of `tsx.opt.6`, outperform even `lock.leave.opt` by approximate 10%.

The HyTM version of SynQuake shows noticeably worse execution time especially in high contention levels. For 1 center quest game scenarios, `hytm` results in a 2.4x slow
down. As explained in our results, the improved transaction commit rates the HyTM version of SynQuake show do not necessarily translate to overall application performance. The automatic privatizing computation phase is rather time consuming as read and write operator overloading, as well as maintaining updated read and write buffers can be costly to the overall execution time. For low contention levels however, we see encouraging results with comparable execution time of the HyTM version similar to the lock based baselines and the TSX based versions.

![Figure 6.15](image)

Figure 6.15: Scalability comparison of all transactional versions of SynQuake against single thread execution running a game with 1 quest at the center of the game map.

### 6.3.4 Scalability Results

Finally, we show the scalability potential of the HyTM version by comparing results measured with different number of threads. We must first note that experiments done on our Haswell hardware platform contains only 4 physical cores and the 8 thread results shown in this dissertation make use of hyper-threaded threads. A hyper-threaded thread shares the same L1 cache with another thread belonging to the same physical core thus can cause unwanted cache overflow aborts when using TSX.

Figures [6.15](#) [6.16](#) and [6.17](#) show the scalability of all baseline, TSX based, and hybrid based versions of SynQuake. We plot the overall execution time of each version against its own single thread performance in order to examine the scalability factor.
Figure 6.16: Scalability comparison of all transactional versions of SynQuake against single thread execution running a game with 4 quests in each of the quadrants on the game map.

Figure 6.17: Scalability comparison of all transactional versions of SynQuake against single thread execution running a game where players moving around randomly with no quests.
For the high contention game scenario shown in Figure 6.15, we note that TSX based versions scale relatively poorly due to the high rate of lock and shared data contention. Running on 4 threads, tsx.opt.0 and tsx.opt.6 only achieved 1.8x speedup. Lock based implementations show similar scaling factors attributing to the high contention for shared data. The most encouraging plot belong to hytm, where using 8 threads with 1 center quest, it can scale up to 4.3x compared to its single thread performance.

For medium and low contention game scenarios shown in Figures 6.16, and 6.17, the lock based baselines and the HyTM version perform comparably when it comes to scaling. Surprisingly, TSX based versions do not show particularly good scaling potential. The limitation for TSX based versions points to the contention for the global single fall-back lock. The HyTM version mitigate and delay such contention throughout transaction execution by introducing additional layers, although the actual execution time still pales in comparison to other versions. We appreciate that our results show scaling potential, although our evaluation cannot easily be extended to a large number of cores until they become available commercially.

6.3.5 Preliminary Optimized HyTM Results

As noted in Section 6.2.2, hytm uses the unoptimized transactions similar to tsx.no.opt and relies on HyTM for automatically optimizations. Additional manual optimizations such as application level code changes are not performed by HyTM. In this section, we show preliminary results of hytm.opt, which contain manually reduced transaction sizes in SynQuake itself in addition to using HyTM as discussed in Section 6.1.6.

Figures 6.18, 6.19, and 6.20 show the transaction composition between all the SynQuake versions with the addition of hytm.opt. Under all game quest scenarios, hytm.opt showed significant increase in the fraction of transactions committed in in HW.Only layer. In the low contention quest scenario, hytm.opt had the greatest impact on HW.Only layer resulting in an increase from 26% to 90% when compared to hytm.

Other than improvements in the fraction of transactions taking advantage of TSX hardware, hytm.opt also shows improved application run-time performance as depicted in Figure 6.21. Under the high contention quest scenario, hytm.opt performs approximately 20% faster than hytm. Under the low contention quest scenario, hytm.opt produces a run-time that is comparable to all of the most optimized lock-based as well as the TSX-based versions.

With these results in mind, we look forward to additional automatic optimizations that can be added to HyTM in the future for further performance improvements.
Figure 6.18: **hytm.opt** transaction composition with 1 quest at the center of the game map.

Figure 6.19: **hytm.opt** transaction composition with 4 quests in each of the quadrants on the game map.
Chapter 6. Evaluation of HyTM Performance

Figure 6.20: **hytm.opt** transaction composition where players moving around randomly with no quests.

Figure 6.21: Execution time of **hytm.opt** running a game with 500 players, 10000 cycles, and under 3 different types of game scenarios.
Chapter 7

Related Work

Transactional Memory (TM) has been the subject of extensive studies in the last decade or so \[30, 14, 15, 36, 12\], but traditionally the focus has been on entirely software approaches, or on hybrid implementations with the hardware part being simulated \[11, 41\]. As actual hardware transactional support has become available only recently from Intel \[20\] and IBM \[16\], new work has been looking at how to successfully couple the existing hardware support with software approaches. However, as far as we know, this dissertation is the first to implement the version based hybrid TM system and evaluate on an actual HTM platform and evaluated with a complex application such as SynQuake. In the rest of this section, we list the many pieces of research we draw inspiration from.

7.1 Hardware Transactional Memory

Wang et. al. \[38\] also investigated IBM’s support for hardware TM; they look at how the STAMP benchmark suite \[7\] 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. While our hybrid solution specifically targets Intel’s TSX platform, we believe the atomicity techniques used in our fall-back layers can be transferred towards other HTM platforms such as IBM’s BlueGene/Q in the future.
7.2 TSX Based Optimizations

Perhaps the closest-related work to our TSX investigation is the very recent study by Yoo et. al. [42] which has been published recently and was conducted concurrent with our work. 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 micro-benchmark (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 in retrying transactions in hardware. As described in Section 5.1 our hybrid TM library allows application developers to optionally configure the number retries allowed for attempting transactions in hardware. A potential improvement on our implementation would be to adjust and tune the retry limits dynamically; Diegues et. al. [13] showcased their work on self-tuning TSX with mixed results. Based on our investigation, adjusting the retry limits does impact overall application performance but eventually they produce diminishing returns. In fact, our previously published work also investigates the effects of retries on TSX execution [39].

Throughout our implementation of TSX experiments and hybrid TM library, we make use of a single global fall-back lock for atomicity guarantee. The reason behind the choice is mainly attributed to ease of implementation, reduced locking overhead, and deadlock avoidance. Optimizations on the single global lock solution have been proposed by Calciu et. al. [6] as well as a more aggressive bloom filter based multiple fall-back lock solution [6]. Despite some of the proposed techniques calling for additional hardware implementation support, their early results show promise to potentially further assist in creating more powerful hybrid TM solutions.

7.3 Hybrid TM Systems

Damron et. al introduced a very thorough design of a complete hybrid transactional memory [11] before HTM system implementations became available to the masses. Dalessandro et. al presented a STM system called NoRec [10] that inspired many iterations
of STM and hybrid TM systems design. Matveev et al. recently presented an hybrid implementation of NoRec [28] that is perhaps the closest to our work in this dissertation. Despite the sizable number of hybrid TM system designs, many of them could not have predicted design constraints and limitations in an real HTM implementation such as TSX and most analysis were done on simulated results.
Chapter 8

Conclusions and Future Work

8.1 Contributions

In this dissertation, 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 parametrized 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 performed comparably to the the best hand optimized fine grain Pthread version. We thus show that high performance is achievable using a set of easy to use APIs, and without the need to use complicated fine-grained locking.

Further leveraging our TSX modelling and profiling findings, we design and implement a new Hybrid TM system called HyTM. It utilizes the TSX based library design from the profiling run and introduce several additional fall-back layers. The added fall-back layers provide automatic variable privatization and transaction size shortening. Using a realistic and complex benchmark called SynQuake, we show that with application specific knowledge from the application developers, HyTM can increase the fraction of transactions successfully committing in hardware by as much as 90%. Despite the overhead and limitations of software based variable operator overloading, SynQuake performed com-
parably to the most optimized lock based implementations under specific game quest scenarios. Lastly, SynQuake running HyTM showed great scaling potential compare to other synchronization approaches and is primed for further studies as platforms with additional core counts become available.

In summary, we make the following contributions in this dissertation:

1. Characterize and model Intel’s Transactional Synchronization Extensions for better understanding of HTM properties, and limitations.

2. Produce a summary of guidelines and recommendations for manual tuning of high performance parallel applications using TSX.

3. Design and implement an original Hybrid TM library providing developers ease of programming and high performing parallel applications.

4. Implement and showcase several benchmarks relevant to HTM and Hybrid TM research including the array access micro-benchmark, and Moldyn.

5. Study HTM and Hybrid TM performance on a realistic application: SynQuake.

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 existing STM techniques such as HyTM. It is our goal to reduce HyTM overhead by eliminate the software based shared access tracking with compiler based optimizations and introduce additional interfaces for more application specific optimization hints. The combination performance and programmability would greatly help programmers towards achieving the ultimate desirable for the programming experience: correctness, simplicity, and performance.

8.2 Future Work

8.2.1 Application Assisted Version Tag Granularity Tuning

The version tags introduced in HyTM has great potential of becoming a key parameter in application-assisted optimizations. It is our belief that parallel applications typically have their own unique characteristics that can impact overall performance in a major way. As described in Section 5.1.2, we intend to use the version tags associated with each shared variable as a unique way of performing conflict detection, resolution and overall correctness guarantee. In our evaluation of HyTM, we statically assigned version
tags in SynQuake in order to reduce RTM transaction sizes, but there is potential to assign version tags dynamically.

In typical parallel applications, it is common to have a load-balancing stage to even the computation load on each computing thread. Stages such as BuildNeighbours in Moldyn and Load_Balance in SynQuake all perform similar tasks. It is during these stages we envision application developers providing help to HyTM. As shared data objects move around such as molecules in Moldyn or players in SynQuake, it is inevitable that they aggregate in some way and form closely related clusters. In such cases, all objects that are in close proximity with each other are more likely to interact compared to objects that are far away. Application developers can optionally hint to HyTM by assigning a common version tag to all of the objects that are close by.

Having shared version tags will not result in major reduction in transaction sizes executing in layer 1 and 4; but they can have a major impact on layers 2 and 3. By having to only record smaller number of shared tags, the size of the hybrid layer’s read-set can potentially be further reduced resulting in a much shortened read-re-validation stage when committing. The downside of having such system would obviously be false-sharing caused by the share version tags.

Investigations are currently on going to leverage application specific features such as load balancing in SynQuake to dynamically adjust version tag granularities.

8.2.2 Application Level Function Tagging

In addition to tag assignments help from the application to reduce transaction size, it is also possible to further reduce the transaction size by tagging functions called from within the transaction as non-transactional. Those functions do not impact shared variables in any way that might compromise atomicity can be tagged by the application developer and in turn ignored by the HyTM system. Doing so allows HyTM to avoid tracking the shared reads and writes done inside these functions (while inside a transaction) and help improve the likelihood of successful hardware transaction commit.

It is worth mentioning that these types of function tags are being discussed as part of the upcoming transactional C++ standard (draft currently, put ref here). While we envision using such system to reduce transaction sizes inside layers 2 and 3 of HyTM, it is also possible that compiler level code transformations with TSX updates can be done to prevent these functions from executing the pure hardware transactions.
8.2.3 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.
Bibliography


[38] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue gene/q


