Using Variable Conflict Granularity to Improve the Performance of Transactional Memory Support for Games

by

Mihai Burcea

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2015 by Mihai Burcea
Abstract

Using Variable Conflict Granularity to Improve the Performance of Transactional Memory Support for Games

Mihai Burcea
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2015

Transactional memory (TM) is a promising parallel programming paradigm for generic applications on large scale parallel architectures. In spite of significant research work in this area in the past decade, adoption by the parallel programming community has been slow due to two reasons. First, performance of most TM support for applications has been disappointing. Second, efforts towards using TM to parallelize realistic applications or highly popular benchmarks for TM have been rare.

It is widely known that two factors can have a drastic impact on the performance of parallel applications with Transactional Memory (TM) support: i) contention among execution threads, particularly contention due to false sharing, and ii) the software TM runtime conflict tracking overhead.

We propose techniques to address application contention and reduce TM conflict tracking overheads by varying the memory access tracking granularity in the TM. To further optimize performance of TM support, we leverage a software transactional memory platform called HyTM based on Intel’s TM hardware support, Transactional Synchronization Extensions (TSX) in Haswell. We build on existing efforts towards developing a gaming benchmark for TM, by characterizing and optimizing the contention patterns in a realistic application, SynQuake, developed based on the open source Quake 3 code.

We port SynQuake to HyTM and design, prototype and evaluate adaptive techniques for varying the conflict tracking granularity of the TM, on the fly, for SynQuake. For regions of the application space causing high degrees of contention, we reduce the tracking granularity while, conversely, we allow a coarse tracking granularity to be used for application regions with low degrees of contention, in order to decrease tracking overhead and transaction size.

Our prototype implements our variable granularity adaptations either by using application-specific knowledge, or, completely transparently, by relying primarily on the support provided by a TM library.

Our evaluation shows that our techniques improve performance by 16.7%, on average, compared to a range of configurations using static parameter settings. More importantly, we show that our techniques are lightweight and can provide statistics that allow both application and user to understand application contention patterns at the cost of negligible runtime overhead.
Acknowledgements

First of all, I would like to thank my supervisor, Professor Cristiana Amza, for her guidance, support, and extreme patience that she’s had throughout the many years it took me to reach this point. I wouldn’t have made it without her constant support, and I am deeply grateful that she had more faith in me than I did myself, at some points in time. I know I wouldn’t have had as much patience with myself in her place. This thesis wouldn’t have existed without her vision and constant efforts.

I am also grateful to my late co-supervisor Professor Greg Steffan, for his advice and his support through these years. His passing away is a great loss to the research and scientific community, and I consider myself lucky to have had the opportunity to work with him.

Many thanks also go to my supervising committee members, Professor Tarek Abdelrahman and Professor Andreas Moshovos, for their patience and efforts to keep me and bring me (repeatedly) on the right track during all these years. The good parts in this work exist partly due to their vision and advice.

I have spent such a long time in grad school, that there is a very long list of friends and colleagues who have helped me in many ways during these years. Their support and help, whether technical, moral, or of any other kind, was crucial in getting me to this point, and I would have likely not made it this far without them. I hope they will forgive me if I omit some of them by mistake.

I owe a great deal of thanks to Daniel Lupei, who was the creator of and main contributor to libTM, a platform that a substantial part of my work is based on. I am grateful for his help and contributions, which lasted long after he finished his degree and left the department.

I similarly owe a great deal of gratitude to Mike Dai Wang, the creator of and main contributor to HyTM, the other platform that my work relies on. Like Daniel, he kept providing support and assistance long after finishing his degree, and he went beyond the call of duty to help me whenever I needed it, always in a timely manner.

My thanks also go to Sahel Sharifymoghaddam for her excellent ideas and insights, and for the brainstorming sessions that have repeatedly helped me improve the quality of my work. Her patience, moral support and help in several other forms have been invaluable. She deserves much more credit than she will ever know or accept.

I am grateful to Ali Hashemi, who always made himself available when I needed it, and was always willing to lend his expertise and help, and constantly provided support - both moral and professional.

My old friend Madhusudan Raman has provided unceasing moral support throughout all these years, (especially) in the hardest of times.

I also owe thanks to Sina Meraji, both for his work and contributions, and for cheering me on and his attempts to boost my confidence.

To all my colleagues-in-suffering through grad school: Chuck Zhao, Utku Aydonat, Davor Capalija, Bogdan Simion, Diego Huang, Zefu Dai - thank you to all of you.
# Contents

1. **Introduction**  
   1.1 Problem Statement .................................................. 1  
   1.2 Contributions .......................................................... 4  

2. **Background**  
   2.1 Speculative Parallelization Concepts .............................. 10  
   2.2 Massive Multiplayer Online Game (MMOG) Concepts ............... 12  
   2.3 SynQuake ................................................................. 13  
      2.3.1 SynQuake Overview .............................................. 13  
      2.3.2 Data Structures ................................................ 14  
      2.3.3 Server Application Structure ................................ 15  
      2.3.4 Quest Scenarios and Game Configuration ................. 15  

3. **Variable Granularity in libTM**  
   3.1 libTM ................................................................. 18  
      3.1.1 libTM Protocol Design ..................................... 19  
      3.1.2 SynQuake Parallelization Techniques with libTM ........ 20  
      3.1.3 Potential of Variable Granularity Tracking for SynQuake with libTM .................................................. 22  
   3.2 Investigating Varying the Conflict Tracking Granularity using libTM .................................................. 23  
      3.2.1 Mechanism for Varying the Tracking Granularity in libTM .................................................. 23  
      3.2.2 Heuristics for Varying Granularity with libTM .......... 25  
      3.2.3 Sensitivity to Parameter Variations of SynQuake using libTM .................................................. 29  
   3.3 Investigation into Contention Sources in SynQuake with libTM .................................................. 30  
      3.3.1 Some Key Data Structures in libTM and SynQuake .......... 30  
      3.3.2 Experimental Investigation into Contention Causes .......... 35  
   3.4 Summary ................................................................. 37  

4. **Intel TSX Characterization**  
   4.1 Intel Transactional Synchronizations Extensions ................. 44  
      4.1.1 HLE vs. RTM .................................................. 44  
      4.1.2 Contention Management ...................................... 45  
      4.1.3 Progress Guarantee and Software Fall-back ............... 45  
      4.1.4 TSX Abort Causes ............................................ 45  
   4.2 TSX-Based Library Design ........................................... 46
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.1 Experimental Framework</td>
<td>116</td>
</tr>
<tr>
<td>7.2.2 The Problem with a Uniform Coarse Grain Approach</td>
<td>117</td>
</tr>
<tr>
<td>7.2.3 Variance in Ideal Granularity</td>
<td>118</td>
</tr>
<tr>
<td>7.2.4 The Potential for Reducing Overhead</td>
<td>120</td>
</tr>
<tr>
<td>7.2.5 Implementation Issues</td>
<td>121</td>
</tr>
<tr>
<td>8  Related Work</td>
<td>122</td>
</tr>
<tr>
<td>9  Conclusions and Future Work</td>
<td>127</td>
</tr>
<tr>
<td>9.1 Conclusions</td>
<td>127</td>
</tr>
<tr>
<td>9.2 Future Work</td>
<td>128</td>
</tr>
<tr>
<td>Bibliography</td>
<td>130</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Transactional Memory (TM) has been widely studied as an alternative paradigm for parallel multithreaded programming. The main goal of TM is to facilitate parallel programming and application development by using transactions. TM replaces the old lock-based synchronization approach with a much easier programming interface: the programmer only needs to specify the begin and the end of regions of code called transactions that will be executed speculatively, in parallel. TM allows transactions on different processors to manipulate shared in-memory data structures concurrently in an atomic and serializable manner. The TM system automatically detects data conflicts and ensures correct parallel execution for generic parallel programs.

The main categories of transactional memory are hardware-based (HTM), pure software-based (STM), or hybrid implementations. There are several popular STM implementations already in use such as: TL2 [20], SwissTM [22], TinySTM [24] and RingSTM [71]. However, STM implementations usually suffer from poor performance due to their significant overheads. The overheads of STM systems come from the need for conflict tracking i.e., the need to track read and write accesses and to determine the conflicts between accesses of different threads at run-time.

As shown on the right hand side of Figure 1.1, a conflict is detected when two threads access the same unit of memory e.g., a word of memory, and at least one of the two accesses is a write. If a data conflict between threads is detected during speculative execution, one or more of the transactions involved are rolled back and restarted.

1.1 Problem Statement

The TM conflict tracking overhead can be decreased by tracking accesses to memory at a coarser tracking unit granularity. For example, instead of tracking memory accesses at the word level as our simple example has assumed, the TM can track memory accesses at the cache-line level, at the 4K page level, or at the arbitrary-sized object level. Tracking accesses at a coarser granularity has the advantage of meta-data aggregation, hence reducing the total memory footprint of the tracking meta-data. This can, in turn, increase performance through improved cache utilization, as well as reduced number of conflict checks.

Unfortunately, increasing the tracking unit granularity comes with an increase in the probability of false conflicts, where conflicting accesses by different threads occur to different parts of a coarse tracking
Figure 1.1: On the left, sequential code is partitioned into regions of code that will be executed speculatively and in parallel, on the right-hand side of the figure. A speculative execution may encounter a conflict between two accesses to the same memory unit e.g., a memory word, which will make a transaction abort and rollback, and then retry execution.

An increase in false conflicts may or may not increase the total number of per-thread conflicts. For example, as shown in Figure 1.2, assume that, when tracking at the word granularity, we register no conflicts between two or more threads on either of two adjacent words, \( w_1 \) and \( w_2 \); however, when tracking at double word granularity, it is possible to register a conflict on the double word consisting of the adjacent \( w_1 \) and \( w_2 \) words when these words are put together in the same tracking unit, say if thread 1 writes the first word, while thread 2 reads both words. In this case, we would experience both true and false sharing when accessing the larger double word unit by the two threads, and no increase in the total number of conflicts when increasing the tracking granularity.

Figure 1.2: The top figure shows true sharing, since two distinct threads are accessing the word on the left side, and one of the accesses is a write. The bottom figure shows conflict tracking at double word granularity; in this case, we experience both true sharing (same as above) and false sharing (due to the \( read_2 \) on the right word and the \( write_1 \) on the left word, which both appear to access the same location due to the double-word tracking granularity). In this case, the false sharing does not increase the number of conflicts, since the accesses by the two threads already experience true sharing.

In general, for a tracking unit granularity \( tu \) of size \( s \), if we increase the tracking granularity by a
factor of $f$, the number of conflicts that would occur in the coarse tracking unit can either be the same or higher than the sum of the number of true and false conflicts that normally occur within each of the $f$ components of size $s$ of the coarse tracking unit. For example, Figure I.3 shows that, if we increase the tracking granularity from word to double word, there would be no increase in false sharing between threads 1 and 2, issuing the $read_1$ and $write_2$ accesses, respectively. If we now double the conflict tracking granularity again to use a 4 word conflict tracking unit, we incur an additional conflict due to false sharing. However, doubling the unit again, from a tracking granularity of 4 words, to a cache line of 8 words, does not further increase the number of conflicts in our example.

Figure I.3: An example of false sharing on a cache line. Two threads are reading, and writing, respectively, distinct portions of the cache line, but since the cache is tracked as one atomic entity, the cache coherence protocol decides that there is a conflict. However, the conflict due to false sharing is the same whether the cache line size is 4 words or 8 words.

An increase in the number of conflicts between transactions is usually associated with increases in abort rates, wait times, live-locks or dead-locks, hence lower performance. Therefore, a fine grained conflict tracking granularity may be conservatively used by most applications, thus missing opportunities for overhead reduction that come with coarser granularities.

For some applications we may be able to predict whether or not increasing the granularity leads to an increase in false conflicts.

Figure I.4: A typical scenario for assigning rows of a matrix to different threads; each thread is allocated three rows; two of those rows are accessed exclusively by each thread, but the third row, located at the boundary between the array regions allocated to each thread, is shared with the other thread.

For example, Figure I.4 shows an application pattern where a two-dimensional array is partitioned into sets of rows, with each set of rows assigned to a different thread for parallel computation. Many classic scientific applications using arrays, such as the Water benchmark from the SPLASH benchmark suite [79] and the FFT benchmark from the NAS benchmark suite [69], use static partitioning as indicated in the Figure, and exhibit both true and false sharing only on elements of the array located at the
boundaries between thread allocations; all other elements are computed on privately by each thread.

In the context of TM parallelization of such array-based code, assume that the set of rows in each partition is computed on in parallel, within a separate transaction. Even if we knew exactly which rows are shared between threads and which ones are not, in all TM systems that we are aware of, the array would be declared as a single TM object, and the TM will track accesses to the whole array at either element granularity or other fixed granularity, such as word or cache line. This would incur unnecessary overheads for tracking accesses to elements placed in rows which the programmer knows to be conflict-free. Ideally, all privately accessed elements of the array within each thread’s partition would be tracked by the TM within a single large tracking unit. There is no API support for this in current TM systems.

In this dissertation, we introduce mechanisms by which the application can group application objects or other entities, e.g., array elements in the previous example, for the purposes of TM tracking as a unit. Moreover, we introduce heuristics for when the above-mentioned mechanisms for coarsening tracking granularity by grouping application objects/entities, versus tracking individual objects/entities should be used. These heuristics take into account either conflict counting in the TM for application entities or conflict prediction for application entities situated at boundaries between thread allocations, by the application.

Even if we cannot always predict which elements situated at the boundary between thread allocations are conflict-prone and which are not, if true conflicts occur in conjunction with false conflicts within a given tracking unit, any increase in false conflicts within a coarse grained tracking unit may not affect performance at all, e.g., in the case where an abort would have occurred anyway. Yet, in other cases, the detrimental effects of increased false conflicts may be offset by the beneficial performance effects of meta-data aggregation. These trade-offs may vary with the application and TM system involved, but, they are usually difficult to explore with existing TM systems.

\subsection{1.2 Contributions}

In this dissertation, we study the effects of varying the conflict tracking granularity in the context of a hybrid software-hardware Transactional Memory prototype, called HyTM, designed and developed in our group \cite{77, 78}, and which is based on the recently released Intel Haswell TSX architectural extensions for TM support. This is, to our knowledge, the first study of the effects of varying the tracking granularity in the context of a hybrid TM, and in particular a hybrid TM leveraging Intel’s TSX Haswell.

Intel TSX is part of recently released hardware TM support. In the past, hardware-based TM systems have mainly been studied through simulations \cite{53, 82} due to the lack of any transactional support in existing hardware. It is only in the past 3 years or so that IBM and Intel released the first systems with TM hardware support, in the IBM Power series, the Blue Gene/Q \cite{23, 31} and the Haswell architecture \cite{37, 83}, respectively.

While most of our thesis investigates the performance trade-offs of variable granularity in the context of tracking conflicts in the software part of our HyTM hybrid, the main advantage of TM systems using hardware support is their significantly lower tracking overhead in comparison with the software-based systems. At the same time, the Hardware TM support usually comes with strict constraints on the transaction size, due to inherent hardware limitations e.g., of cache size, the effects of exceptional control flow on cache functionality, etc.

The conflict tracking granularity for hardware transactions in Intel TSX is the cache line. Specifically,
the cache coherence protocol of the L1 cache is used to provide support. This currently limits the size of the transactions that can be effectively supported to the size of code that can be run without any cache replacements or invalidations, since a cache replacement or invalidation would automatically cause a transaction abort.

In order to enable us to run non-trivial applications in our study of variable granularity for TM, we implement and evaluate heuristics for dynamically varying the consistency granularity in the context of HyTM, instead of using the baseline TSX Hardware TM. HyTM increases the probability of transaction commit by collecting the read and write accesses in a software transaction and atomically flushing the collected writes in hardware at commit, through a TSX-supported hardware transaction. Besides flushing the write accesses at commit, for correctness, HyTM needs to perform a read validation of all read accesses collected a-priori in software, in the same hardware transaction commit.

Therefore, for the purposes of our study of variable granularity, HyTM offers opportunities for studying the performance trade-offs involved with different tracking granularities in software, and their side-effects on the associated hardware transactions. As mentioned previously, such side-effects of increasing the software tracking granularity include a reduction in the meta-data footprint in the cache, hence potentially fewer hardware aborts due to cache replacements. Further advantages of using coarser granularities in software for HyTM are potential reductions in the number of read validations performed in the hardware transaction at commit. However, as also mentioned, all these advantages come at the cost of potentially an increased number of aborts during read validation due to false conflicts and the trade-offs between these factors are application- and platform-dependent.

We explore the trade-offs involved in varying the conflict tracking granularity in the context of SynQuake, a game benchmark modeled after Quake, which was previously developed in our group [46]. This benchmark shares most of its data structures with the open source first-person-shooter Quake 3 game [36], which was used as a basis for developing the benchmark. We believe that the use of realistic applications as benchmarks for Transactional Memory is important for the TM community. In spite of the ease of use claim for the TM programming environment, and the availability of the STAMP TM benchmark suite from Stanford since 2008, we believe that TM benchmarking efforts have been rare, and no benchmark has yet gained enough popularity to have reached standard status for TM experimentation and evaluation. This is especially the case for benchmarking efforts for leveraging Hardware TM support in Haswell.

We build on the initial SynQuake benchmarking efforts mentioned above [46], which were focused mostly on outperforming lock-based SynQuake. While we do not outperform lock-based SynQuake with HyTM SynQuake, we provide a comprehensive analysis of performance sensitivity to various application and TM parameters, thus characterizing the sources of contention in the SynQuake game with our Haswell-based TM Hybrid.

Moreover, in our prototype implementation of techniques for varying the conflict tracking dynamically in HyTM running SynQuake, we introduce a minimally intrusive way for the application and TM to collaborate and exchange information at run-time. Specifically, in our enhanced version of HyTM, the TM can inform the application about the number of conflicts that each application object causes to running transactions. The application can use this information in order to change the TM tracking granularity for its objects. Moreover, the tracking granularities used by the TM are chosen as units of spatial locality that are meaningful in application space. Thus, the application can use its semantic information about object co-location in application space, its object-to-thread allocations and interactions
between objects in order to predict conflict-prone objects and modify their tracking granularity even before conflicts occur.

Figure 1.5: A typical situation in a multiplayer game: players have a limited area of interest, as their movement and visibility are usually restricted to nearby regions and objects.

For example, Figure 1.5 shows typical assumptions used in games which need to be taken into consideration when assigning objects to threads for load balancing the work across threads. A player’s avatar has limited movement speed and sensing capabilities. In the Figure, the area a player avatar can sense and impact through player actions, called area of interest, is illustrated by the shaded area surrounding the player. Items and players outside this area cannot be detected or acted upon by the avatar. This means that data access in games exhibits both temporal and spatial localities. Games and real-time simulations have exploited this property and applied interest management to game state. Interest management allows us to limit the amount of state any given player has access to to the area of interest of that player. Thus, if we assign different regions of spatial locality in the game to different threads for the purposes of load balancing, a player might impact game state in neighboring regions because its sight and range of action extends across region boundaries.

Therefore, players that are located close to boundaries between regions that belong to different thread allocations in application space, i.e., game map, cannot cause conflicts for their respective transactions by interacting with each other and with other objects on the game map unless they are within a distance from each other that is lower than the area of interest. Although access patterns are much more dynamic in games than in the typical array-based scientific computation, the sharing pattern just described shares similarities with the array-based access pattern introduced earlier and offers similar opportunities for conflict prediction and refinement of the tracking granularity for player objects in the game.

Thus, assuming that the load balancer uses spatial locality in the game in its assignments of objects to threads, we can use coarse tracking granularity for objects that fall within a thread’s allocation and fine granularity to track objects close to the border of thread allocations, as before.

Because the access patterns in games are more dynamic, and also access meta-data in a shared
manner, however, we use several different tracking granularities for SynQuake, not just two. We leverage a tree meta-data already used in Quake, hence in SynQuake, in order to augment the application meta-data with TM statistics associated to TM conflicts on both application data and meta-data. The tree is a spatial-search data structure used for storing and locating objects within the game map. The interface between application and TM remains the standard lean TM interface, consisting of transaction begin and end markers, with the addition of giving the application the ability to manipulate the TM tracking granularity of its own objects. This is performed through an additional function that maintains the mapping between each application object and an associated TM-maintained object, called an object tag, which tracks TM statistics such as the application object's number of writes in the TM and number of conflicts in the TM. All objects sharing the same TM-maintained object tag will be tracked as a unit for the purposes of read validation, incrementation of the version number of the group upon each write, and conflict tracking. While the grouping of the objects can be arbitrary, our heuristics use object co-location in application space i.e., on the game map for grouping objects.

In this environment, the coarsening of TM tracking granularity of application objects is equivalent to sharing the mappings of an increasing number of co-located application objects to one TM object tag. The refinement of TM tracking granularity is the opposite i.e., decreasing the number of co-located application objects sharing the same TM object tag.

We believe that this pattern of interactions between application and TM is novel compared to all previous TM systems that we are aware of. In previous TM platforms, the inner workings of the TM have normally been kept transparent to the application. In these previous TM systems, the TM is normally used to track conflicts at a static, predetermined unit size, or rely on compile-time analysis to decide what the ideal granularity should be for various data structures [49]; in contrast, our adaptation benefits from knowledge available at run-time and can dynamically adjust granularity during the execution of the application.

Moreover, in all previous TM systems we are aware of, TM conflict tracking had little or nothing to do with application semantics such as object access patterns and object location in application space. While the work by Atoofian [4] proposes a run-time approach for varying the TM granularity, their investigation does not clearly describe the heuristics or mechanisms used and proposes to vary granularity only for contiguous memory addresses. In contrast, our approach can group together multiple independent application objects for tracking at a coarser granularity, and is based on either application or TM run-time knowledge.

In our performance evaluation, we first show that, because of limitations in the TM support in Haswell, while the Haswell-based TM support promises high performance, this performance is highly sensitive to various parameter settings in the TM. In this part of our evaluation, we present results of profiling two applications on Haswell, an array-based micro-benchmark and SynQuake, which demonstrate that these applications need to be modified by hand in order to be able to take advantage of the hardware TM support at all. Although tedious, these manual code modifications improve the fraction of hardware transaction commits, hence performance. We further show that, with our hybrid TM, it becomes possible to improve performance for the SynQuake application even more, through leveraging hardware for supporting the commit phase of longer software-hardware transactions.

Finally, we also show that adding our techniques for variable tracking granularity to the hybrid TM support for SynQuake can further improve the performance of the game with negligible programming effort, by a factor of 16.7%, on average, across the different configurations used in our experiments.
More importantly, in our experiments, our variable granularity heuristics do not add any measurable overhead, regardless of game scenario and starting parameter settings.

In terms of baselines for comparison, we first show the performance of the game while conducting manual off-line parameter configuration profiling with SynQuake on HyTM. SynQuake has many application parameters such as the types of game actions allowed, the number of obstacles and consumables in the game, the number of players, the size of the game map, the player speed, support for multiple quest scenarios, and so on.

We conduct extensive off-line profiling, including with different TM conflict detection and resolution parameters and different application-informed heuristics, specifically by using a know-it-all algorithm for granularity assignment that is aware of quest locations and durations.

Currently, our techniques for varying the tracking granularity on the fly provide performance improvements in many configurations, although static profiling and the use of know-it-all application-informed algorithms can still outperform our dynamic variable granularity algorithms in certain configurations.

While more experience with commercial game scenarios or N-Body simulations may provide further insights, we suspect that other simulation codes use a similar data structure as SynQuake for placing and searching objects efficiently on a 2D or 3D game map, and have a similar pattern of quiescent periods between server processing of consecutive frames during which the game map and tree meta-data is updated. We believe that this may be the case, because this type of tree data structure and access patterns were used in the Barnes N-Body simulation benchmark from the SPLASH suite. If this is the case, then our tree meta-data augmented with HyTM statistics can provide support for parallelizing other applications or developing new ones based on our skeleton framework.

In addition, we show that the implementation of similar heuristics for variable tracking granularity can be ported to SynQuake running with a different, fully software TM system, called libTM. libTM was previously shown to outperform locking on the same game application we use - SynQuake [44, 46].

We also observe that other applications with good scalability prospects on TM platforms may exhibit similar features and access patterns as our game, specifically coarse-grained sharing between threads, and application programming with spatial or temporal locality in mind. Such examples might be applications that use R-trees [28] or spatial-temporal database query processing [27].

While in our game, spatial locality refers to object co-location on a game map, some of the same concepts apply if memory accesses to objects have spatial and temporal locality in terms of adjacency of memory addresses. We show that an implementation of our conflict prediction heuristic which we introduce for the game can be ported to a classic scientific kernel with regular access patterns on arrays, called Successive-over-relaxation, SOR, which does not use any special meta-data, or application maps. It is therefore reasonable to believe that in other TM applications, where each thread is allocated a relatively coarse-grained data partition, with object accesses in each data allocation for each thread expected to exhibit both spatial and temporal locality, and conflicts occurring only between objects located at the boundary of thread allocations our conflict prediction heuristics are applicable as well.

Finally, we show, through an emulation of a speculative framework using the SPECint benchmarks that, if full support for varying the conflict tracking granularity across the full range of word-level granularity, several cache-line sizes, and page level granularity ever became available in hardware, standard scientific benchmarks would likely be able to take advantage of it. Specifically, we show that the ideal tracking granularity varies across all SPECint benchmarks as well as across various code regions of each benchmark, where we define the ideal granularity as the largest granularity where no additional false
conflicts appear compared to all available finer granularities, for each of the studied speculative code regions.

In summary, the contributions of this dissertation are as follows.

- We extend an existing software-hardware hybrid TM system developed in our group (HyTM) with support for varying the tracking granularity, and with heuristics to dynamically vary the access tracking granularity at run-time.

- We characterize the sources of contention in a realistic benchmark SynQuake using HyTM.

- We conduct an evaluation which shows that our enhanced HyTM system with support for variable access tracking can reduce TM overhead for SynQuake by minimizing false conflicts in areas of high contention, while reducing tracking overheads and transaction sizes.

- We study the potential of porting our heuristics for variable granularity to other speculative parallelization frameworks and to other applications. We also perform a potential study for providing full hardware support for varying the conflict tracking granularity for scientific applications in speculative parallelization frameworks.

The structure of the remainder of this document is as follows: we begin by providing background information on terminology and concepts that will be used throughout this document in Chapter 2; in the same Chapter, we also introduce our main application, SynQuake. In Chapter 3 we introduce our software TM library, libTM, and we present our investigation into the effects of variable granularity in the context of libTM. Chapter 4 describes the basics of the Intel hardware TM support, and our baseline hardware TM library built on top of Intel’s hardware TM support. We present our hybrid TM framework, HyTM, in Chapter 5. We explain in detail the mechanics of our implementation and show our experimental results in Chapter 6. In Chapter 7 we show our findings on varying granularity in conjunction with other TM systems and applications, and we study the potential for providing full hardware support for varying the conflict tracking granularity. Finally, we discuss related work in Chapter 8 and wrap up with conclusions and future work in Chapter 9.
Chapter 2

Background

In this Chapter, we introduce some of the field-specific terminology and define several terms that will be used throughout this document. We then present the main benchmark that we use throughout our experimental evaluation, SynQuake, which is a simulator for a Quake-like Massive Multiplayer Online Game (MMOG).

2.1 Speculative Parallelization Concepts

The abundance of computational units in modern hardware has led to a continuously-increasing popularity and usefulness of parallel programming. With this also come attempts to find parallel programming models that provide ease of programmability and good performance at the same time. Two of the most well-known such parallel programming paradigms are Thread Level Speculation (TLS) \cite{29,72} and Transactional Memory (TM) \cite{34,56,68}.

Both TLS and TM come in hardware, software, or hybrid flavours. The bulk of our experimental results were achieved on a hybrid hardware and software TM system called HyTM, and is focused on varying the tracking granularity in software. Therefore, we focus most of our introductory material on transactional memory concepts. A potential study described in Section \ref{sec:potential} was performed on a TLS framework.

TM allows transactions on different processors to manipulate shared in-memory data structures concurrently in a data-race-free manner. TM replaces traditional locking synchronization in parallel programs with a much simpler programming interface. Instead of explicit fine-grained locking of data items, the programmer specifies the beginning and end of parallel regions with transaction delimiters. A runtime TM system (based on a software library linked with the program) automatically detects data races between concurrent transactions and ensures correct parallel execution for generic parallel programs. Any detected incorrect execution resulting from a data race causes one or more transactions to be rolled back and restarted. The run-time system automatically detects which memory regions are read and written by a transaction, and maintains the recoverability of data for the written ranges of memory.

While TLS and TM have most of their main features in common, such as dependency violation tracking and detection, buffering of results, checkpointing and replaying, for the sake of completeness, we briefly describe some of the differences between TM and TLS.
TLS systems usually rely on partitioning a program into ideally non-conflicting tasks, commonly determined by high-level programming language constructs like loops; these tasks are speculatively executed, and inter-task violations are detected and tasks rolled back and re-executed in such cases. In contrast, TM systems rather relax the strict exclusion of speculative code regions as enforced by traditional lock-based programming [5, 84].

Types of conflicts A conflict occurs when two or more threads try to access the same shared resource at the same time, and at least one of the two accesses is a write. There are three types of such conflicts:

- **Read After Write (RAW)**, also known as a true conflict or dependency: this occurs when one thread reads the shared data after the other one wrote to it.

- **Write After Read (WAR)**, also known as an anti-dependency, occurs when one thread or transaction writes the shared data after the other thread read it.

- **Write After Write (WAW)**, also called an output dependency, takes place when both threads attempt to write to the same shared resource.

Conflict Detection and Conflict Resolution Speculative parallelization systems have two main ways to manage situations caused by the conflicts presented above. These are, respectively, conflict detection (deciding when to detect that a conflict has occurred) and conflict resolution (deciding what to do after a conflict was detected).

Conflict tracking or detection is the process of tracking memory accesses to identify and resolve dependences, determine whether transactions should commit, and ensure atomicity: if multiple concurrent transactions access the same memory location and at least one access is a write, then there is a potential conflict which can be resolved by letting one or more of the transactions abort and retry. By "concurrent" we mean that neither transaction starts after the other one commits.

We will briefly mention some of the more common approaches, but a detailed discussion of the tradeoffs of these methods is beyond the scope of this document, and can be found in related literature sources [51].

The conflict detection policy determines when conflicts are detected: this can happen eagerly (when the conflict occurs), which results in a pessimistic policy, or lazily (at commit time), which leads to an optimistic policy. These policies can be mixed for different conflict types.

The conflict resolution policy decides what to do in the case where a conflict was discovered; common approaches are Polite (where the transaction that discovered the conflict always waits for the other transaction to finish), Aggressive (the transaction that discovers the conflict always aborts the other transaction). In terms of WAR conflicts, these can also be viewed as Wait-for-Readers or Abort Readers, respectively.

Maintaining Shared Writes There are two main strategies for maintaining writes to shared memory locations: Write-Buffer and Undo Logging.

Write Buffering [60] requires each thread (transaction) to maintain a private buffer for any writes made to shared data; upon a successful commit, the write buffer is flushed to memory in a single atomic operation. The main advantage of using this technique is that aborts are cheap, since we only need to clear the write buffer (which can be done as easily as resetting an index to zero).

With Undo Logging [73], writes are always performed to main memory, but a log is maintained of any such changes made to shared data. In case of an abort, the log needs to be parsed and all changes
reversed to the original state they had before the start of the transaction. However, commits are cheap with this approach, since the changes have already been applied to memory.

**True Sharing and False Sharing** "Sharing" refers to concurrent accesses to a part of an object or an entity; *true sharing* means that the concurrent accesses are actually on the same part of the object; *false sharing* means that the concurrent accesses are on different fields of the object, but to an external observer that can only tell if the object was modified or not, it appears that the accesses modified the same thing (the object itself). These concepts are very common when tracking granularity is done at the level of cache lines: if two threads modify different locations of the same cache line, false sharing takes place; while no sharing between threads actually occurs, the (undesired) effect is as if true sharing has occurred: the cache line is considered "dirty" and needs to be updated, just as if the two threads had tried to modify the same location. This is because the cache coherence protocol is not aware of changes within a single cache line - namely, the coherence protocol sees a cache line as a unit; in effect, the protocol can perceive changes in the cache at the granularity of a cache line, but no finer than that. In this dissertation, we will refer to conflicts or violations that occur due to false sharing as false conflicts, or false violations, respectively.

**Tracking Granularity** True and false sharing are deeply connected to the concept of tracking granularity. TM and TLS systems need to track reads and writes during speculative regions, in order to determine when concurrent accesses occur on the same shared variable.

Tracking granularity refers to the unit size at which these accesses are tracked. For example, an intuitive approach would be to track each variable independently. However, all variables are not equal: does it make sense to track an entire (large) array as a single entity? If we did, then *any* accesses to the array would be considered a conflict that could abort a transaction, even if completely different parts of the array had been accessed. Therefore, perhaps it would be better to track accesses to this array with a finer e.g., element granularity. Going to the other extreme, we could track each element of the array independently. This would eliminate any false conflicts, but would incur a significant overhead due to the very large number of entities that we are now tracking.

In summary, for TM, conflict tracking is the process of tracking memory accesses to identify and resolve dependences, determine whether transactions should commit, and ensure atomicity: if multiple concurrent transactions access the same memory location and at least one access is a write, then there is a potential conflict which can be resolved by letting one or more of the transactions abort and retry. Tracking memory at a very fine granularity provides very good accuracy, but has high tracking overhead. In contrast, coarse-grain conflict tracking has much less tracking overhead, but can suffer from expensive false-conflicts.

What an "ideal" granularity would be (and some methods for achieving it in a dynamic and adaptive manner), is the core topic of this dissertation.

### 2.2 Massive Multiplayer Online Game (MMOG) Concepts

Massive Multiplayer Online Games (MMOG) are multiplayer games where one of the main attractions lies in the large number of players that participate in the game. MMOGs represent a multi-billion dollar industry, with one of the most well-known of them, World of Warcraft, having had more than 12 million active subscribers at its peak and still having over 7 million active subscribers as of May 2015 [7]. Some MMO games have a persistent universe, where all players are logged into the same virtual world; Eve
Chapter 2. Background

Online is one such example, with a record of more than 65,000 players logged in at the same time [12].

Our main benchmark is a realistic MMOG simulator, SynQuake [46], described in detail in Section 2.3. We now describe two key concepts in MMOGs, both of which are also present in our own application.

Flocking  Flocking [10, 81] refers to the situation where a large number of players gather in a small area of the world. Two of the quest scenarios we use in our experiments express this phenomenon.

Interest Management  The concept of Interest Management [81] refers to the fact that players usually have limited movement and visibility capabilities, and therefore are only likely to interact with objects that are close to them. This concept represents part of the framework that can allow a game to benefit from variable granularity access tracking: groups of players that are close together could be tracked at coarse granularity without concern for false conflicts, if they are unlikely to interact with players managed by different threads (a server thread may be responsible for processing a group of players that can be chosen based on various criteria, such as player location in the game world). On the other hand, if players are close to areas where there are players managed by different threads, then it may make sense to use fine granularity to track those players. This concept can be looked at as spatial locality in the context of the game world.

2.3 SynQuake

In order to evaluate our hybrid TM 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 [36]. Unlike popular TM benchmarks such as STAMP [55], SynQuake provides more realistic TM usage scenarios and a wider range of flexibility in parameters.

In this section we present our benchmark, SynQuake, and some of the details of our experimental setup. We also describe in detail the parallelization efforts, transactional region composition, and optimizations performed.

2.3.1 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.


2.3.2 Data Structures

The SynQuake game world is comprised of different types of entity objects: players, apples, walls, and potentially others. Player objects include many data fields: player position, player attributes such as health level, score, player movement, 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 experiments we eliminate expensive graphical computations such as explosions and game world changes to isolate the effect of parallel execution.

In order to achieve shorter transaction duration that has a chance to commit in hardware, in our experiments with SynQuake we use no walls and player actions are reduced to player moves. In contrast, in the original SynQuake work, more game physics i.e., collisions with walls, and complex actions e.g., move and eat, or move and fight were used, which allowed libTM to outperform a lock-based implementation of SynQuake \cite{46}.

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 \ref{fig:area}). 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.

2.3.3 Server Application Structure

![Figure 2.2: SynQuake client processing stages](image)

Building on the knowledge of the game’s previous developers, the SynQuake server code consists of three stages as shown in Figure 2.2. These three stages are request processing, administrative tasks, and reply processing. The server loops through these three stages forming a server frame or iteration. In the first stage, the server receives requests from clients. These client requests get processed by a worker thread, each worker thread being responsible for handling their own particular clients. The thread assignments are created according to load balancing policies and are updated in the administrative stage. Load balancing is the periodic reassignment and redistribution of workloads to different threads, with the goal of obtaining roughly equal workloads for each thread. The second stage or administrative stage as seen in Figure 2.2 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 4.3.1).

2.3.4 Quest Scenarios and Game Configuration

As a TM research benchmark, the SynQuake server application uses a large set of configuration parameters. These parameters allow 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 (the situation where a large number of players gather to a small part of the world) 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.

Figure 2.3: SynQuake player distribution with 1 quest in the center of the map.

Figure 2.4: SynQuake player distribution with one quest in the center of each of the 4 quadrants on the game map.

Quest locations in SynQuake can impact the contention level of the game.

We use three different scenarios to simulate different contention levels and how they impact TM
behaviour as well as overall application performance. We use the following different quest scenarios:

1. A scenario with one quest in the center of the map, where all players flock towards, shown in Figure 2.3 henceforth referred to as the 1qc scenario. This is a realistic quest scenario that simulates flocking [81].

2. A scenario with four quests, one for each game map quadrant. Players flock towards the quest location closest to them, shown in Figure 2.4 which we will refer to as the 4qq scenario. This would be a different case of flocking.

3. A scenario with no quests, where all players move around randomly, shown in Figure 2.5 from now on referred to as the noq scenario.

**Load Balancing** In addition to quest scenarios, load balancing algorithms also impact overall game server performance [46]. However, load balancing is beyond the focus of this study, and while SynQuake does support several dynamic load balancing algorithms, for our experiments we limit the load balancing algorithm to only static3, which statically associates specific regions of the map to threads (e.g., one quadrant to each thread).
Chapter 3

Variable Granularity in libTM

In this Chapter, we introduce our variable granularity extensions to our software transactional memory library, libTM. In Section 3.1 we give an overview of libTM and some of its features and mechanisms. We then present our investigation into varying the tracking granularity in libTM in Section 3.2. In Section 3.3 we present a discussion, supported by experimental results, of the contention sources in SynQuake with libTM, and we summarize our findings in Section 3.4.

3.1 libTM

libTM [14, 46] is a highly-customizable STM developed in our group that supports multiple conflict detection and conflict resolution policies.

In a TM program supported by the libTM library, shared data needs to be distinguished from private per-thread data accessed inside transactions. For this purpose, transactional shared and private variables should be declared using the meta-types \texttt{tm\_shared} and \texttt{tm\_private}, respectively. For example, a shared variable \texttt{int x} in the original program needs to be declared as \texttt{tm\_shared\<int\>x}. The definition of each of these meta-types in the libTM library is a C++ template using the original type of the variable as a parameter, (e.g., \texttt{tm\_shared\<original\ type\>}).

Similarly, all objects that will be accessed inside a transaction have a specific data type declaration, which is a wrapper around the base type of the object (plus additional metadata). This is necessary for tracking all accesses to these shared objects, and maintaining read sets and write sets for transactions, and it is accomplished by operator overloading for these data types.

Specifically, any read or write accesses on shared and private transactional variables are tracked inside the implementation of the overloaded conversion or assignment operators.

Furthermore, libTM maintains recovery data for both \texttt{tm\_shared} and \texttt{tm\_private} variables updated in transactions, while performing conflict detection and resolution only for \texttt{tm\_shared} variables.

A more detailed description, as well as a sample listing of an overloaded operator can be found in Section 5.1.1 and Listing 5.1 respectively.

The application programming interface (API) for the library consists of several easy-to-use macros that allow thread creation and destruction, parallel execution, as well as indicating boundaries for transactional regions. The API has significant overlapping parts with the API for HyTM (which was based on libTM), and we provide more detailed descriptions of these macros when we introduce HyTM.
3.1.1 libTM Protocol Design

While the TM protocol used by libTM can be customized, the main protocol component that is shared for all libTM protocol variants represents a variation of the classic two-phase locking concurrency control algorithm, resulting in a blocking implementation of transactional memory semantics.

In all libTM versions except its most optimistic version (fully optimistic) conflicts are solved according to the two-phase locking protocol. Deadlocks are detected and broken using time-outs on waiting for locks.

In its fully optimistic version, libTM solves write-write conflicts by maintaining multiple private copies of a shared object, although locks still need to be obtained in the commit phase in order to apply the concurrent writes to shared memory in the order of committing transactions.

Any read-write conflicts detected at are resolved by either aborting the conflicting reader transactions or waiting for the reader transactions at commit. Accordingly, there are two policies called wait-for-readers and abort-readers.

Our libTM library uses an invalidation strategy that relies on visible-readers. Specifically, every reader records its access of a memory location in the visible-readers set associated with that memory location. Consequently, performing a read also involves a write with this policy. An updating transaction sets the abort status of all reading transactions for the updated locations before committing. To avoid any inconsistent executions, a transaction checks its abort status at every operation on shared state.

3.1.1.1 Conflict Detection in libTM

libTM features multiple conflict detection and resolution policies, and we extend all of these with support for dynamic variable-granularity conflict tracking. The conflict detection methods determine when locks are to be acquired for any shared data (such as a player) during a transaction. Four conflict detection policies are supported:

**Fully Pessimistic**  Both read and write locks are acquired at access time.

**Partially Read-optimistic**  Write locks are acquired on access, readers wait if a writer exists, otherwise readers proceed optimistically without locks.

**Read-optimistic**  Write locks are acquired on access, readers always proceed optimistically without locks.

**Fully Optimistic**  Write locks are acquired at commit time, readers always proceed optimistically without locks.

More pessimistic transactions are more likely to succeed, but produce more lock contention. Conversely, fully-optimistic transactions will only hold locks for a very brief interval at commit-time, minimizing contention but increasing the chance of aborts by detecting any potential conflicts too late—which the only solution is to restart the transaction. Since our goal is to optimize the granularity at which locks are acquired, in principle, more-pessimistic policies have a much larger potential for improvement due to their higher lock contention.
3.1.1.2 Conflict Resolution in libTM

libTM also supports two conflict resolution policies: *wait for readers* and *abort readers*. These determine how readers who attempt to access a location modified by a transaction are treated. The conflict detection and conflict resolution policies are fixed parameters for a given execution; they are among the main factors that are affected by locking granularity and have a significant impact on performance.

3.1.1.3 Rollback Mechanism in libTM

Transactional memory updates become final only when a transaction commits. Consequently, the libTM library needs to implement some mechanism for ensuring that uncommitted writes are not final. Since the main conflict detection policy is optimistic, using write-buffering represents the only viable option. With write-buffering all the writes are performed in a private buffer and the shared data is updated only in case of a commit. To avoid read-after-write hazards, the write-buffer needs to be searchable so that a thread is able to locate and correctly read data that it has previously written in the current transaction. For improved performance, libTM ensures searchability by using a hashtable and a Bloom filter.

3.1.2 SynQuake Parallelization Techniques with libTM

In this Section, we describe the overall approach taken for parallelizing SynQuake with transactional support provided by libTM. The parallelized SynQuake version we use is based on a previously parallelized version of the SynQuake game with lock-based synchronization. The code changes performed were simply converting lock-based 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 2.3.3, two transactions are identified: one for adding clients who join the game, and the other for processing client actions. In this work, we focus our efforts on the transaction that processes client actions, as it runs many times throughout an entire game session and is the source of the majority of contention in the application.

The transaction that processes client actions performs a list of tasks inside the critical section; these tasks are 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 (AN) and determine whether the current player’s action range rectangle overlaps with the AN player’s current position area rectangle.
4. If the player checks all the players that are close by and determines that his movements can be performed without landing on any existing player’s current position, the move is performed. Otherwise, the player is moved to a position without conflict in the direction of movement.
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 this work, we focus our investigation on transactions where the player action to be processed involves player movement where the player action is constrained only by the placement of other players on the game map; no obstacles are used on the game map. Listing 3.1 shows the pseudocode for player movement in SynQuake.

A bounding box for a player move is computed conservatively as a function of the player’s position, direction, and velocity of movement. Figure 3.1 shows the scenario where a collision may occur between two moving players. If the bounding boxes of the player moves overlap, depending on the players’ speeds and the distance between them, a collision between players may occur. When a collision occurs, the effective distance a player moves will be less than the distance the player could potentially move in the absence of the collision. When a collision with another player’s position on the game map is detected, the player is assumed to stop just short of the collision position, in their direction of movement. If players are situated in direct proximity of each other, the computed distance of movement of a player may be zero. If the distance computed is zero, then the player does not perform a move.

Listing 3.1: Player actions in SynQuake with libTM.

```
BEGIN_TXN
for each player p1 {
    compute compound range for all possible actions
    if action is a move {
        set player speed, direction
        compute potential moving distance distance_to_move
    }
    compute action range rectangle
    collect nodes of interest from whole tree {
        overlapping_nodes = list of nodes that the
        action range rectangle would intersect
    }
    for each node in overlapping_nodes list {
        for each player p2 in that area node {
            if action ranges of players p1 and p2 overlap
                if p1 and p2 can actually reach each other
                    with current speeds and distances
                    then {
                        // collision, p1 may only move up to the collision point
                        distance_to_move = distance_between (p1, collision_point)
                    }
        }
    }
}
if distance_to_move != 0 {
    actually perform the player move {
        if player p1 moves across nodes {
            delete player p1 from current node
            add player p1 to new node
        }
        update coordinates of player p1
    }
}
END_TXN
```
Figure 3.1: Player bounding boxes overlap. If players can potentially reach each other given current speeds, there will be a collision; in that case, the players will only move up to the collision point (and possibly not move at all, if they are already next to each other).

3.1.3 Potential of Variable Granularity Tracking for SynQuake with libTM

In this Section, we first explain the trade-offs between using different tracking granularities in libTM and we motivate variable granularity adaptations at run-time for SynQuake in Section 3.1.3.1; we then present the tracking granularities we use in libTM in Section 3.1.3.2.

3.1.3.1 The Performance Trade-offs of Conflict Tracking Granularity

Our previous study of a parallelization of SynQuake using libTM [46] showed that the transactional version of the game outperformed locking at the largest (8-thread) configuration. At the time, we believed that this was due to reduced data contention in the transactional version of the game compared to the locking version of SynQuake. However, the sources of contention and, in particular, the relationship between contention and the tracking granularity in the TM were not fully investigated in our previous work. In this Section, we study the relationship between tracking granularity and contention in libTM.

Data contention can be defined as the attempt of multiple threads to access shared data items at the same time. Since correctness requirements force these accesses to be serialized, the negative impact on performance can be significant. Contention can arise because of both true and false sharing.

True sharing occurs in the game when multiple players that are assigned to different threads are within each other's area of interest and interact with each other, or compete for an entity that is on (or very close to) the border between the game regions covered by those threads. If we consider only moves as possible interaction types, then a move causes true sharing if the bounding boxes of two or more players overlap. In the case of overlap, while attempting to perform a move for one player, the server reads all other player's positions that are relevant to the overlap; however, those positions read may later become invalid within the same transaction, if those other players succeed to move to other positions. This is an example of true sharing caused by contention on the player positions.

False sharing occurs either by itself or in conjunction with true sharing when the conflict tracking granularity is larger than the immediate bounding box(es) of the objects affected by each interaction in the game. Hence, it is generally known that a finer conflict tracking granularity may reduce contention.
due to false sharing, while a coarser conflict granularity reduces the tracking cost. On the other hand, as we can infer from our previous illustration of the player interactions caused by player moves, other factors that affect the range of player actions, such as player density, player speed, and the allocation of players to threads, may play a role as well.

3.1.3.2 Conflict Tracking Granularities in libTM

Our software transactional memory library, libTM, has support for different conflict tracking granularities: word, object or entity, set, and thread. We have extended libTM by adding support for variable granularity - that is, the ability to independently change granularity for different objects or groups of objects dynamically, during runtime. Before our extension, during a given execution of the program, all player objects used the same granularity, and this granularity could not be altered throughout the execution of the program. In the default conflict tracking granularity, i.e., word level granularity, conflicts are detected based on the meta-data information encapsulated in each tm\_shared object.

In this study of variable granularity in the context of libTM, we use only two granularities: fine or object granularity i.e., entity-level, and one coarser granularity i.e., set-level.

If entity-level tracking granularity is used to track an object, then each object is tracked by libTM as a unit. For the purposes of tracking at the coarser set-level granularity, all objects that are stored in each leaf node of the area node tree are tracked as a unit, while objects that span the boundaries between two leaf nodes are tracked at the higher granularity of the closest common ancestor of both leaves. This means that the finest granularity at which libTM can track reads and writes to player coordinates is determined by the region of the game map represented by an area node leaf, which also corresponds to the grid unit in the game.

3.2 Investigating Varying the Conflict Tracking Granularity using libTM

In this Section, we describe our implementation for support of variable granularity in libTM and present several heuristics to evaluate variable granularity in SynQuake with libTM.

We first present the mechanism for varying granularity in Section 3.2.1 in Section 3.2.2 we describe several heuristics for SynQuake, in increasing order of application knowledge; we study these heuristics in conjunction with varying the conflict detection and conflict resolution policies in libTM, in Sections 3.2.2.1 through 3.2.2.4. We conclude with an experimental evaluation of these heuristics with SynQuake in Section 3.2.2.5.

3.2.1 Mechanism for Varying the Tracking Granularity in libTM

Changing the tracking granularity in libTM on the fly is done through a mechanism that can map (and remap) the meta-data of a tm\_shared object to point to a TM-maintained meta-data object. Since the main conflict detection protocol in libTM is based on two-phase locking, in order to have access to any data within an object e.g., a player object, a thread has to first acquire a lock protecting this object. For this purpose, each object contains a lock pointer to the respective lock, and changing the tracking granularity is accomplished by redirecting the lock pointer to point to a coarser grain lock.
Chapter 3. Variable Granularity in libTM

24

Figure 3.2: Variable-granularity conflict tracking in libTM. The left-hand side of the picture shows an area node tree, with each node maintaining a list of objects located in its corresponding area on the game map. A player in that list can have its lock pointer point to the entity-level within the player object, or to the set-level lock that exists in every area node. On the right-hand side of the picture, we show how within the same grid unit, some players are tracked at entity granularity, whereas others at set granularity.

The implementations of entity-level and set-level granularity are depicted in Figure 3.2. If entity-level tracking granularity is used to track an object, then the lock pointer of that object points to a lock field within the object itself, referred to as “entity-level lock” in Figure 3.2.

For the purposes of tracking at the coarser i.e., set-level granularity, several objects can be grouped into a set by redirecting their respective lock pointers towards a shared lock. For set-level granularity, we use the lock field inside the area node that contains the respective objects as the shared lock, referred to as “set-level lock” in Figure 3.2.

Once a player object is tracked in this manner at the set-level, it will continue to be tracked at this level, no matter its location on the game map. When a player moves from one area node to another, its lock pointer is updated to point to the set-level lock of its new area node. Therefore, within each game region corresponding to an area node, we may have a group of players tracked as a set, as well as players tracked individually, as entities, as shown in the right-hand side of Figure 3.2.

The set granularity we use is the grid unit in the game. All accesses to the objects within a grid unit are performed through the respective area node (usually a leaf node), and can be protected by the area node lock. Hence, changing the tracking granularity within the corresponding game area can be done correctly, on the fly, during the administrative single-threaded stage between server processing of two consecutive frames. While we do not explore this aspect in this Chapter, tracking granularities coarser than grid unit can also be implemented by using a non-leaf area node lock as the set-lock.

By remapping the lock pointer of several semantically-linked tm_shared objects to the same shared lock object, we can effectively increase the granularity of the access tracking, and by that, reduce the overheads associated with bookkeeping inside the libTM library. This approach offers several benefits when compared to the alternative of static word-level or object-based granularity. First, remapping can take place at runtime and can factor in semantic aspects that may vary in time. Secondly, it can handle dynamic data structures as well as variables that aren’t colocated in memory next to each other, whereas objects need to be statically determined at compile time.
3.2.2 Heuristics for Varying Granularity with libTM

In this Section, we introduce four heuristics for SynQuake and we study them in conjunction with varying the conflict detection and conflict resolution policies in libTM. We present the four heuristics in the order of using increasing degrees of game state knowledge.

3.2.2.1 Reactive Density-based Adaptation

This adaptation technique is a simple algorithm that decides the conflict tracking granularity by looking at the density of players contained within each region of the game map corresponding to a grid unit i.e., area node leaf.

If the number of players within a grid unit is above a high density threshold, then we adapt to entity granularity for conflict tracking of that region of the game map. In this case, a thread accessing any player within that grid unit will acquire its corresponding entity-level lock. For lower player densities, set granularity is used for that grid unit of the game map. In this case, the locks for all corresponding players will point to their area node’s set level lock. The intuition behind this technique is that a high player density in a region of the game map could be conducive to false sharing and a high number of conflicts within that region.

3.2.2.2 Reactive Conflict-based Adaptation

Our reactive conflict-based adaptation uses the number of actual inter-thread conflicts within each grid unit i.e., area node leaf, rather than the player density for triggering granularity adaptations.

The observation is that, regardless of the player density, conflicts occur only when two or more threads are competing for ownership of the lock of the same player. This technique pays the cost of periodically traversing the area node tree and counting actual conflicts between different threads occurring within each grid unit i.e., area node leaf.

If all players in an area node belong to the same thread, then there are no conflicts in interactions involving these players; we group these players to be tracked at the set granularity of the area node. Otherwise, if the number of actual inter-thread conflicts is above a threshold, we switch to entity tracking granularity for all player objects within that grid unit.

3.2.2.3 Proactive Conflict Predictive Adaptation

The reactive approach to contention presented above waits until conflicts actually occur, and then it changes the granularity of the players involved, if appropriate. Therefore, contention due to false sharing may degrade performance before adaptation is triggered.

We can address this limitation by proactively predicting conflicts, and changing the tracking granularity before it may become necessary to do so. Specifically, by default, we use coarse (set) granularity in the STM for all grid units of the game. When a player approaches the boundary region with the game region assigned to a different thread, we proactively switch the locking level to fine (entity) tracking granularity for that player.

3.2.2.4 Proactive Quest-aware Adaptation

With this heuristic, we assume that the STM runtime has access to near-perfect knowledge of quest appearance or disappearance, quest location, and the approximate delay interval for players to disperse
when a quest becomes inactive. These are all game artifacts that the game designer would be aware of, or would be able to provide estimates for.

In this adaptive technique, while quests are known to be active, we use a fine (entity) tracking granularity in the STM. After quests disappear, the STM allows for a dispersal delay interval, and then switches to a coarse (set) granularity.

### 3.2.2.5 Experimental Evaluation

In this Section we show an experimental evaluation of our adaptive techniques in the context of libTM. We start with the quest-aware adaptation to show the maximum benefit we can get using near-ideal game knowledge in the STM. We then show the performance of our other adaptive techniques. We also compare the performance of our adaptive techniques against static granularity baselines.

The experiments presented in this Section and in the entire Chapter were performed on a machine with four 2-way Hyperthreaded CPUs at 2.8GHz with 16GB of RAM, running a 2.6 Linux kernel and gcc 3.3.6. Threads are bound to processors such that each thread executes alone on its (Hyperthreaded) CPU. Each result represents an average over three runs. We manually inspect the results for any significant variance or outliers.

Game simulation is split into equal-duration quest sessions. Within each quest session, quests are inactive for the first half then active for the second half. We ensure that quest sessions are sufficiently long that players have time to both gather near active quests and disperse from inactive quests. The result is that the game has periods of high contention when quests are active and players gather around the quests, and low contention when quests are inactive and players disperse. The level of contention can be controlled via the number and placement of quests on the game map.

We study two different quest configurations, $1qc$ and $4qq$, as described in more detail in Section 2.3.4.

We use the execution time of a fixed number of server frames in our game simulation as the most representative metric. From a player’s perspective, this metric should indicate the responsiveness of the server, which is directly related to the quality of the gameplay experience.

**Proactive Quest-aware Adaptation** Figures 3.3(a) and 3.3(b) present the performance of three approaches: fixed granularity, and near-perfect knowledge of quest appearance and disappearance with two player dispersal delay intervals. The delay intervals correspond to the players’ dispersal from around a quest location once the quest disappears, when a fine granularity is still beneficial. We show two different delay intervals because, due to randomness of player movement and speed, it is difficult to accurately estimate the time it takes players to disperse sufficiently for a coarser granularity to become preferable.

Figures 3.3(a) and 3.3(b) show two things: first, they confirm that the fixed granularity approach is clearly suboptimal, and it can be improved on; second, we get an accurate estimation of the maximum performance we can expect, assuming perfect knowledge of changes in the game state.

In the following, we show the impact of our other adaptive techniques while varying the STM conflict detection policy. We expect our adaptive methods to have a greater impact for the more pessimistic conflict detection policies. This is because our algorithms adjust the granularity for each player with the goal of keeping conflicts as well as true or false sharing to a minimum. For optimistic policies, locks are acquired late (at commit time), and therefore held for a minimal amount of time; this in turn means that there is much less potential for conflicts (since threads are less likely to compete for the same lock at the same time). In contrast, for pessimistic policies, locks are acquired earlier and held longer, which
Chapter 3. Variable Granularity in libTM

(a) Comparison between a fixed granularity approach and near-perfect knowledge of quest cases for the 2 quest scenarios. The three leftmost bar groups in each figure show data for entity granularity; the other three for set granularity. For each granularity, the first bar group shows results for switching granularity upon quest on/off (MS), switching granularity 250 cycles after quest on/off (MS250), and fixed granularity (Fixed). For each bar in a group, we show the conflict detection (RO = read-optimistic, FO = fully-optimistic) and conflict resolution (A = abort readers, W = wait for readers).

makes it more likely for conflicts to occur. As our methods target conflicts, and because there are more conflicts for pessimistic policies, our algorithms are more likely to be able to improve performance in such cases. Our results shown in the following paragraphs confirm this expectation.

Reactive Density-based Adaptation We show our evaluation for the density-based approach in figures 3.4(a) and 3.4(b). We evaluate the impact of this method versus fixed entity and set granularities, using thresholds of 5, 10, 15 and 20 players. The performance of our algorithm follows closely that of the ideal static granularity (entity) in the first scenario; it fares less successfully in the second scenario, where it still mirrors the performance of fixed entity granularity, although in this case set granularity would perform better. However, even in this case, our algorithm stays within 10% of the ideal fixed granularity.

Reactive Conflict-based Adaptation The performance of the conflict-based approach is presented in Figures 3.5(a) and 3.5(b). In the 1qc scenario, the heuristic stays within 1% of the performance of the best static granularity on average. For the 4qq scenario, the algorithm’s performance stays within 1% of the performance of the ideal static granularity, with two scenarios where our approach outperforms static granularity by 20% and 26%, respectively.

Note that the missing bars in Figure 3.5(a) and Figures 3.6(a) and (b) represent parameter configurations for which execution did not complete successfully: the program timed out. More specifically, the library detects deadlocks and breaks them by aborting the transactions involved; however, these deadlocks, followed by the subsequent transaction aborts, can keep occurring repeatedly, in which case the program effectively makes no progress. We handle these scenarios by explicitly aborting an experiment that is still running after a user-defined timeout period.

Proactive Conflict-Predictive Adaptation Figures 3.6(a) and 3.6(b) show the performance of the conflict-predictive, location-based method. This algorithm takes a proactive approach to contention for players owned by different threads and tries to prevent contention from occurring in the first place, by switching to entity granularity for players that are considered likely to enter into such conflicts.
Figure 3.4: Performance of density-based approach for the 2 quest scenarios. For each group of bars we show the conflict detection (RO = read optimistic, FO = fully optimistic) and conflict resolution policy (A = abort readers, W = wait for readers). Each bar within a group shows performance for a specific threshold: 0, 5, 10, 15 and 20 (a threshold of 0 means fixed granularity).

Figure 3.5: Performance of conflict-based approach for the 2 quest scenarios. We show the conflict detection policy (PO = partially read-optimistic, RO = read-optimistic, FO = fully optimistic) for each successive pair of bars, and the conflict resolution policy (A = abort readers, W = wait for readers) for each bar. Bar height above zero on the Y axis means speedup with respect to when h2 is turned off; bar height below zero on the Y axis means slowdown.
Figure 3.6: Performance of the location-based approach for the 2 quest scenarios. For each successive pair of bars, we indicate the conflict detection policy (PO = partially read-optimistic, RO = read-optimistic, FO = fully optimistic), and the conflict resolution policy (A = abort readers, W = wait for readers). Bar height above zero on the Y axis means speedup with respect to when h3 is turned off; bar height below zero on the Y axis means slowdown.

Our algorithm stays within 1.5% of the performance of any fixed granularity approach for the 4qq scenario. No significant improvements occur, but this is expected, because the positioning of the quests (far from areas owned by other threads) does not cause the heuristic to kick in for most players. However, it is important to observe that our method does as well as any combination of conflict detection, conflict resolution, and fixed granularity.

For the 1qc scenario we stay within 5% of the performance of fixed granularity configurations, and we show significant improvement in two cases, of 16% and 26%, respectively. The reason for improvement is because the fixed granularity in these cases is set - which naturally performs poorly when there is a single quest that is on the border of each thread’s owned area. Our approach detects this and adjusts granularity accordingly to a finer grain, leading to the increased performance.

3.2.3 Sensitivity to Parameter Variations of SynQuake using libTM

In our presentation of our work in this Chapter, we have so far assumed that varying the granularity of grouping players in the game would have a significant performance impact. Based on this hypothesis, we implemented and evaluated 4 heuristics, which we presented in Section 3.2.2. As we have seen, our experimental results were mixed. Therefore, in this Section, we extend our exploration of contention in the SynQuake game through a sensitivity analysis to other game parameters that we have not studied so far.

We conduct an off-line profiling examination of the application by varying the following application parameters: area node tree depth, and the number of players. We keep constant other parameters such as the player speed and the size of the game map. We use the same quest scenarios as described in Section 2.3.4.
3.2.3.1 Varying Contention by Varying the Area Node Tree Depth

Varying the depth of the leaf nodes in the area node tree associated with the game map automatically varies the size of the grid unit on the game map. Larger grid unit sizes likely translate into increased data contention for a given number of players because of potential false sharing effects. Game objects that span grid unit boundaries always induce higher contention because they are placed in non-leaf nodes. However, in our evaluation we vary the tree depth of leaf nodes, hence implicitly the number of players in the leaf nodes, therefore the level of both meta-data contention and data contention in the game.

The experiments shown in Figures 3.7, 3.8 and 3.9 were performed with 2000 players, over 8000 cycles, and on a map of size 512x512.

Figures 3.7, 3.8 and 3.9 show that finding a "sweet spot" for the tree depth for different game quest scenarios is difficult: this sweet spot seems to be around a tree depth of 10 or 11 for the 1qc scenario, but a tree depth of 8 for the noq scenario.

Furthermore, Figure 3.10 shows that when varying the number of players, again the "ideal" tree depth varies widely: for lower numbers of players (below 250), a very shallow tree (depth of 4) yields the best performance. However, as the number of players increases, a tree depth of 10 seems to be the best option.

Based on these experimental results, we argue that the performance of SynQuake running on libTM is sensitive to the variation of various parameters of the application with a direct impact on contention. Another take-away point of these figures is that finding an ideal sweet-spot for a parallel application with multiple parameters with TM support is non-trivial.

From the sensitivity analysis presented in this Section, we noticed that SynQuake performance is much more sensitive to the variation of the SynQuake area node tree depth than to the granularity of the player grouping or lack thereof (entity/set). We next present in Section 3.3 a more extensive investigation into the sources of contention in SynQuake in order to explain these trends that we observed experimentally.

3.3 Investigation into Contention Sources in SynQuake with libTM

In this Section, we present a detailed investigation into the sources and types of contention in libTM, and provide explanation for various performance aspects of SynQuake running over libTM. While this Section refers to libTM, the patterns of contention and performance are similar to the ones experienced with HyTM as well, and libTM and HyTM share similar key data structures; therefore, most of the explanations in this Section apply to HyTM to the same extent.

We begin by presenting key data structures and explaining their impact in Section 3.3.1 and then we show a series of experimental results that shed more light on specific performance aspects in Section 3.3.2.

3.3.1 Some Key Data Structures in libTM and SynQuake

As explained in Section 2.1 and Section 3.1.3.1 in more detail, there are two types of conflicts: true conflicts, and false conflicts. True conflicts occur when the same actual data is being accessed, while false conflicts take place because the memory access tracking granularity is too coarse.
Figure 3.7: Execution times for different fixed tree depths for the 1qc scenario. In each bar group, the first bar represents ent granularity, while the second bar is for set granularity.

Figure 3.8: Execution times for different fixed tree depths for the 4qq scenario. In each bar group, the first bar represents ent granularity, while the second bar is for set granularity.
Figure 3.9: Execution times for different fixed tree depths for the noq scenario. In each bar group, the first bar represents ent granularity, while the second bar is for set granularity.

Figure 3.10: Execution times for different fixed tree depths and different numbers of players for the noq scenario. Timings are normalized with respect to the execution time for the fixed tree depth of 10. This experiment shows the results for fixed ent granularity.
Figure 3.11: Execution times for different fixed tree depths and different numbers of players for the noq scenario. Timings are normalized with respect to the execution time for the fixed tree depth of 10. This experiment shows the results for fixed set granularity.

In the case of SynQuake running over libTM, both conflicts occur, in different amounts, with different parameter configurations. We will discuss the fully optimistic conflict detection policy, as this is the policy used in most of our libTM experiments in this dissertation.

We will also pay particular attention to the Wait-for-readers conflict resolution policy, as it exhibits some counter-intuitive behaviour that can explain some performance trends.

Listing 3.2: Partial description of the hashtable data structure in libTM.

```c
typedef struct tm_hashtable_t: public tm_obj {
    tm_type<tm_list_t*> bkts;
    tm_int n_bkts;
    tm_int n; //number of elements in the hashtable
    tm_int fx; //used for resizing the hashtable
    ...
} tm_hashtable_t;
```

We begin by presenting two key data structures: Listing 3.2 shows the code for the hashtable that is a part of every area node in the area node tree; this hashtable contains the list of players present in that area node, more specifically, the list of players that are on the map region that corresponds to a specific area node. The smaller the tree depth, the larger the map area that each node covers. Conversely, for deep area node trees, each node covers a small part of the map, and will likely contain a relatively small number of players (except in cases of flocking).

An important point about the hashtable is that all its fields are tm_type objects, that is to say, they are transactional objects. As such, they are always locked when accessed during a transaction. However,
the key point here is that all but one of the fields in the hashtable share the same unique lock, that is used for the entire hashtable. The \( n \) field, representing the number of elements in the hashtable, uses its own lock for set granularity only, and shares the same lock with all the other hashtable metadata fields.

Listing 3.3: Partial description of the player data structure in libTM.

```cpp
class tm_entity2 { 
public:
    tm_element e;
    tm_rect r; //position; cause of true conflicts
    vect_t size;
    ...
    int ent_type;
    int ent_id;
    tm_value_t attrs[0];
}
```

The second key data structure we present is the player data structure in SynQuake, shown in Listing 3.3. The structure is 148 bytes large, which is larger than the size of a cache line (64 bytes). The most important field in Listing 3.3 is \( \text{tm_rect}\ r \), which represents the player position. This is updated every time the player performs a move with a distance greater than zero.

The consequences of these data structures are as follows: true sharing conflicts cause a bottleneck on the hashtable, for each insert or delete operation. Every single write access to the hashtable causes the unique hashtable lock to be acquired, essentially serializing the access to the hashtable. In such cases, false sharing, such as incurred by too-coarse granularity, is overshadowed by true sharing.

In addition to this true sharing on meta-data caused by the hashtables, true sharing on game data also exists; this occurs on the \( r \) field in the player object, representing the player’s position, and only takes place in the case of successful player moves. In high-contention scenarios such as \( \text{qc} \), successful player moves are rare due to player collisions, as all players are crowded around the central quest. In low-contention scenarios such as \( \text{noq} \), successful player moves are much more frequent, and thus conflicts due to updates to player position will have a bigger impact.

Writes to player positions will increase conflicts due to false sharing on cache lines; when \( \text{ent} \) granularity is used, there will be no increase in conflicts and no false sharing due to granularity, but there will be an increase in false sharing when \( \text{set} \) granularity is used.

Furthermore, in libTM relatively few conflicts turn into aborts. Aborts only happen for deadlocks in the commit stage with the Wait-for-readers conflict resolution policy; the pseudocode for the commit stage in libTM is shown in Listing 3.4. The key point here is that with the Wait-for-readers conflict resolution policy, the writer waits for all readers to finish before acquiring write locks. In high-contention scenarios such as \( \text{qc} \), there are many readers, as all players are bunched up in the center of the map, around the single quest. Therefore, despite low abort rates, performance will be poor in such cases, due to the very long waiting times for all the readers in the commit stage.

Looking at each individual quest scenario now, for the \( \text{qc} \) scenario, the main bottleneck is not the fraction of writes, but the very large number of readers, and our Wait-for-readers policy. The data structure bottleneck is represented by the insert and delete operations on the hashtable, caused by the players who still move (at the edges of the crowd).

For the \( \text{noq} \) scenario, the main bottleneck is the hashtable again, since player moves are common,
Listing 3.4: Pseudocode for the commit phase in libTM for fully optimistic conflict detection with Wait-for-readers conflict resolution.

```plaintext
acquire locks {
    wait_for_readers();
}
if conflicts or deadlocks detected
    abort;
flush writes from write buffer;
release locks;
```

and players cross area node boundaries very frequently.

In the 4qq scenario, there is a combination of the patterns described for the other two quest scenarios, but conflicts are minimized due to ideal load balancer partitioning for this scenario, as there are no inter-thread conflicts once the players gather around the four quests, since the distances between the quests are large enough such that player bounding boxes only include players that are owned by the same thread.

### 3.3.2 Experimental Investigation into Contention Causes

In this Section, we show a series of experiments that will help explain some of the performance aspects of SynQuake running with libTM.

All experiments presented in this Section were run with 1000 players, for 4000 cycles, on a map of size 1024x1024, and all results represent the average over three runs.

Figures 3.12, 3.13 and 3.14 show the execution times for the 1qc, 4qq and noq quest scenarios, respectively. Figures 3.15, 3.16 and 3.17 show the abort rates for the same quest scenarios, while Figures 3.18, 3.19 and 3.20 show the level 2 cache misses for the three quest scenarios.

We note that due to high contention, the experiment timed out for set granularity, for a tree depth of 4, for Wait-for-readers. For the same parameters but with a tree depth of 8, the application did not time out, but experienced high variance in execution time. These outcomes are caused by deadlocks and associated repeated aborts; the deadlocks are in turn caused by high contention, as we explain in Section 3.2.2.5.

The bowl-shape trend of execution times is perhaps best explained by Figures 3.18 and 3.19, for a low tree depth, we experience a significant number of cache invalidations. These cache invalidations cannot be caused by false sharing on the cache lines, because due to the size of a player object, the position of two player objects cannot be on the same cache line (the size of a player object is 148 bytes, while the size of a cache line is 64 bytes). Some false sharing may occur due to set granularity, but not at the cache line level.

We observe from the Figures that the trends in execution time of all experiments are generally correlated with the trends in abort rates and with the trends in the number of cache misses, respectively. Specifically, higher abort rates and higher number of cache misses typically correspond to higher execution times in almost all experiments.

In more detail, our belief that the level of contention in the 1qc scenario is higher than the level of contention in the 4qq scenario correlates with the overall abort rate trends in the two scenarios. Specifically, the abort rates registered for all experiments in the low contended 4qq scenario are negligible.
(below 0.01%). In contrast, the abort rates in 1qc reach values above 25% in 4 experiments and even reach 100% in one experiment. In this latter case, repeated transaction abort and replays cause the experiment to time out. Because all conflict detection protocol variants in libTM are based on two phase locking, aborts occur only in the case of deadlocks for all experiments using a wait for readers conflict resolution policy. From this data, we draw two conclusions:

First, the main source of contention in most (all but four) experiments is not the abort rate per se, but the wait times required by the wait-for-readers conflict resolution policy and also the overheads and timeout periods induced by the deadlock detection algorithm in libTM. Second, in several 1qc experiments, and especially in the four experiments with 1qc where the abort rate is significant, the underlying reason is repeated deadlocks upon transaction replay; even when deadlocks are detected and broken, and the respective transactions are replayed, no forward progress is made for some transactions because the pattern repeats itself. In libTM, there is no provision for eventually running transactions serially in order to resolve such cases; therefore, as mentioned, experiments can time out as a whole, and not finish successfully.

While the pattern of repeated deadlock and transaction replay can, in principle, cause some unpredictability in behavior, we observed that this pattern occurs quite consistently in 1qc with shallow area node trees; we also observed in experiments presented earlier in this Chapter that repeated deadlocks with no progress patterns are more likely to occur for the more pessimistic conflict detection libTM protocols. Furthermore, as shown in the Figures, the repeat deadlock pattern is more likely to happen with a tree depth of 4 rather than 8, and with a granularity of set rather than ent, because the true sharing on the tree metadata is highest at low tree depths, and this true sharing is compounded with false sharing on game data in the set configuration.

We further notice that increasing the tree depth to 14 is correlated with marked decreases in abort rates in all experiments, including the 4 pathological experiments in 1qc with high abort rates due to repeated deadlocks just mentioned. Moreover, the abort-readers conflict resolution policy appears to be more effective than wait-for-readers in addressing high contention. This is reflected in lower execution times and abort rates when this policy is used instead of the default wait-for-readers conflict resolution policy. This confirms the earlier discussed inference that the detrimental performance impact of contention occurs mainly due to high wait times and deadlocks rather than due to wasted work during transaction abort and replay. Of particular interest is the fact that waiting for readers in cases of conflict causes higher abort rates in the highly contended 1qc scenarios than aborting the conflicting reader transactions. While surprising at first glance, if we consider a case where a writer transaction may have to wait for thousands of concurrent reader transactions, as is the case with 1qc with thousands of concurrent players (in our case 1000 all within interaction range), this clearly brings an elevated probability of repeated deadlock, hence more aborts and no progress for any transaction. In contrast, if we abort all concurrent readers upon detecting a conflict, at least the writer transaction will successfully commit, hence forward progress is more likely for some player moves.

Finally, we also observe that the experiments with high deadlock-induced abort rates also register a high number of cache misses. For the pathological cases in 1qc, this is clearly due to executing the higher number of total operations of the repeated transaction deadlock, abort and replay pattern. Conversely, we notice that, for 4qq with a high tree depth of 14, we encounter a very low abort rate coupled with peaks in the number of cache misses relative to the 4qq experiments with other tree depths. The reason for this trend in 4qq is the high memory footprint of the accesses to tree nodes on the pathways to
accessing leaf nodes in the area node tree causing thrashing of the cache due to capacity misses.

On the other hand, we notice that the total numbers of cache misses in the 1qc experiments with the same tree depth of 14 is similar to the respective numbers in the 4qq experiments (approximately 400K cache misses total in the 1qc experiments versus around 350K in the 4qq experiments), while the effect of contention in 1qc is still dominant for shallow trees. Specifically, we notice that the effects of contention in 1qc manifest in increased cache misses for the lowest tree depth of 4 in 1qc, where the cache misses vary between 500K and 800K, compared to the total cache misses of 400K with a tree depth of 14 mentioned above. These additional cache misses in cases of high contention are not due to capacity misses; they are also only partly due to repeated deadlocks. We suspect that write accesses to the visible readers set, which is a libTM data structure shared between threads, may cause significant cache invalidations, therefore significant misses due to invalidations and cache line thrashing in cases of contention.

In summary, we have inferred that the effects of meta-data and data contention manifest due to true sharing on tree meta-data, particularly on the hash tables maintained in the area nodes in all 1qc configurations with shallow trees; this is coupled with false sharing on game data in the respective set granularity configurations. We have also inferred that the high contention pitfalls of shallow trees manifest as potentially high abort rates due to repeated deadlocks and also high cache miss rates due to executing more instructions and due to cache invalidations on writes to the visible readers set. Even in cases of low abort rates, the wait for readers conflict resolution policy can induce higher execution times than the abort readers policy due to long waiting times, which, in turn may result in higher number of overall aborts due to deadlocks. In contrast, the pitfall of deep trees is due to a high memory footprint which causes a high number of cache capacity misses hence an increase in execution time.

### 3.4 Summary

One of the important parameters that affects the performance of TM implementations is the granularity of conflict tracking. In this Section we have described the way we extended libTM with support for variable granularity conflict tracking, to allow dynamically adjusting granularity for transactional objects during program execution, as well as independently tracking different entities with different granularities.

Since it is unlikely for a user to be able to guess the best performing static granularity for a given set of parameters, we study various approaches: based on player density, the type of conflicts, player location, and quest location. All optimization techniques are applied to a parallel implementation of a Quake-like MMO game, SynQuake, using a software transactional memory, libTM. We showed that our approaches perform as well as the best static granularity in most cases, and perform up to 26% better in specific scenarios.

We choose to focus our further investigation and evaluation of variable granularity adaptations on our hybrid hardware-software TM library, i.e., the HyTM platform, for two reasons: i) HyTM holds the promise of (significantly) higher performance than libTM due to leveraging hardware support for TM, and ii) as we will show in Chapter 5 in Section 5.4, HyTM has even higher sensitivity to transaction sizes and configuration parameters than libTM, hence the potential performance impact of our adaptations is higher.

In general, we believe that leveraging hardware support for TM will have increasing impact in the future, particularly on larger, more powerful multi-core architectures than Haswell, and our investigation
on Haswell may provide early insights into the use of automatic adaptations to enable TM applications to match or even outperform their lock-based counterparts at lower programming effort.
Chapter 3. Variable Granularity in libTM

Figure 3.12: Execution times for the 1qc quest scenario. We show results for both conflict resolution policies, Wait-for-readers and Abort-readers, for both entity and set granularities, and for three area node tree depths: 4, 8 and 14.

Figure 3.13: Execution times for the 4qq quest scenario. We show results for both conflict resolution policies, Wait-for-readers and Abort-readers, for both entity and set granularities, and for three area node tree depths: 4, 8 and 14.
Chapter 3. Variable Granularity in libTM

Figure 3.14: Execution times for the noq quest scenario. We show results for both conflict resolution policies, Wait-for-readers and Abort-readers, for both entity and set granularities, and for three area node tree depths: 4, 8 and 14.

Figure 3.15: Abort rates for the 1qc quest scenario. We show results for both conflict resolution policies, Wait-for-readers and Abort-readers, for both entity and set granularities, and for three area node tree depths: 4, 8 and 14.
Figure 3.16: Abort rates for the $4qq$ quest scenario. We show results for both conflict resolution policies, Wait-for-readers and Abort-readers, for both entity and set granularities, and for three area node tree depths: 4, 8 and 14.

Figure 3.17: Abort rates for the $noq$ quest scenario. We show results for both conflict resolution policies, Wait-for-readers and Abort-readers, for both entity and set granularities, and for three area node tree depths: 4, 8 and 14.
Figure 3.18: L2 cache misses for the 1qc quest scenario. We show results for both conflict resolution policies, Wait-for-readers and Abort-readers, for both entity and set granularities, and for three area node tree depths: 4, 8 and 14.

Figure 3.19: L2 cache misses for the 4qq quest scenario. We show results for both conflict resolution policies, Wait-for-readers and Abort-readers, for both entity and set granularities, and for three area node tree depths: 4, 8 and 14.
Figure 3.20: L2 cache misses for the noq quest scenario. We show results for both conflict resolution policies, Wait-for-readers and Abort-readers, for both entity and set granularities, and for three area node tree depths: 4, 8 and 14.
Chapter 4

Intel TSX Characterization

In this Chapter, we present the details of Intel’s Transactional Synchronization Extensions (TSX), a baseline TM library with hardware TM support based on Intel TSX, and our parallelization strategy for SynQuake to enable leveraging TSX support.

4.1 Intel Transactional Synchronizations Extensions

Intel Transactional Synchronizations Extensions (Intel TSX) \cite{39, 83} 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.

4.1.1 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. \textit{XACQUIRE} and \textit{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: \textit{XBEGIN} and \textit{XEND}. An additional instruction \textit{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.
4.1.2 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 [67].

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 [35]. 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 [67], we identify TSX’s conflict detection policy to be pessimistic (eager) and conflict resolution policy to be abort-readers. By deploying such policies, TSX can maintain a 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.

4.1.3 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 to 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 [40].

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 one software transaction (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 4.2.3.

4.1.4 TSX Abort Causes

TSX transactions utilizing both HLE and RTM can be aborted due to a variety of reasons [16]. In addition to real shared data conflicts, TSX-enabled transactions can easily be aborted due to various other reasons, as listed below:

1. Real data conflicts as described in 4.1.2,

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 4.1.2, TSX tracks shared data at cache-line granularity.

4. Conflicting accesses to the global fall-back lock: as described in 4.1.3.

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 by reading the status codes in the EAX register \[16\], and they are used throughout our library design for code path changes.

### 4.2 TSX-Based Library Design

TM systems are complex in nature and can be comprised of many individually designed components. In order to ensure consistency, our library implementations makes 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.

#### 4.2.1 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 4.1.1, 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.

All results shown represent the average over three runs. We manually inspect the results for any significant variance or outliers.

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 \[80\]; however they are beyond the scope of this dissertation.

#### 4.2.2 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 \[44, 46\] 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 topics.

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 \textit{parallel_func} with a predefined thread count; it takes the \textit{parallel_func} function pointer to any function in the application, the id of the thread executing it, and any other \textit{arg} as arguments. \textit{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 where and under what circumstances a transaction has begun. Such information is used extensively in our specific Hybrid library implementation in order to perform the necessary roll-back actions on aborts.

In addition to the two sets of APIs described, there is also another set of interfaces for tracking shared variable reads and writes. As explained in Section 4.1.2, 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 later on in Chapter 5.

### 4.2.3 Software Fall-back Path Implementation

As described in Section 4.1.3, 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 still a topic of current research; we have chosen the simplest form as recommended by Intel: with a global fall-back lock [16]. 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 4.2.2, 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 4.1 and Listing 4.2 show the pseudo-code transformations for our implementation of the transactional API.
Figure 4.1: TSX with simple software global lock fall-back (FB) path.

To further assist in understanding the code path a transaction might take, Figure 4.1 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 4.1.3, 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 4.2 line 6).

4.2.4 Case Studies: Array Access Micro-benchmark and SynQuake

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. We also describe the modifications performed to get SynQuake to make better use of hardware transactional resources. 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 4.3, 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-
Listing 4.2: RTM hardware transaction implementation with single global fall-back lock.

```c
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);
  }
```


```c
ArrayAccessBenchmark() {
  GenerateAccessPattern();
  for every iteration in N iterations {
    BEGIN_TRANSACTION();
    // reads and writes based on access pattern
    AccessArray();
    END_TRANSACTION();
  }
}
```

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.

2. **Contiguous Mode**: Each thread accesses `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 `write_ratio`.

### 4.2.4.1 TSX Modelling Results

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
Chapter 4. Intel TSX Characterization

Figure 4.2: 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 4.3: Abort rate vs. txn size vs. write ratio for the array access micro-benchmark using 4 threads, 0 transaction retries, and contiguous access pattern.

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.

Our hardware platform is the one described in Section 4.2.1.
Next we present a representative subset of our characterization results.

Figure 4.2 shows a 3 dimensional plot of abort rates vs. transaction size vs. write ratio for random access patterns. For read-dominated transactions, the abort rate becomes very high when 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 4.4: Throughput 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 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 reveal that the overhead of entering and exiting a RTM transaction can be as much as 600 nanoseconds on our hardware platform. The RTM overhead is amortized in larger transactions, thus revealing a peak in overall transaction throughput.

Finally, Figure 4.6 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.

4.2.4.2 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:
Figure 4.5: Throughput vs. txn size vs. write ratio for the array access micro-benchmark using 4 threads, 0 transaction retries, and contiguous access pattern.

Figure 4.6: Abort rate vs. write ratio and the number of retries for the array access micro-benchmark with contiguous accesses, 4 threads.

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 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.

4.3 Experimental Setup

In this Section, we first introduce some of the optimizations we performed on SynQuake, and then we present our experimental environment and benchmark setup details.

Our experiments are run on an Intel i7-4770 Haswell chip-set with TSX support, and are compiled with GCC v4.8.1. Our hardware platform is the one described in Section 4.2.1. All of the results presented are averaged over 3 runs to eliminate the effect of possible statistical errors and run-time anomalies. All experiments use 4 threads without hyper-threading.

Using the SynQuake benchmark described in Section 2.3 we use three quest scenarios to simulate different contention levels. We describe these quest scenarios in more detail in Section 2.3.4.

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.

4.3.1 Manual SynQuake Transaction Optimization

As described in Section 3.1.2, the client action processing transaction performs many actions. Having such long transactions can easily cause RTM transactions to overflow and abort. Aside from introducing a baseline version with TSX support, we also implement versions with manual optimizations for transaction size reduction in SynQuake in order to be able to have a reasonable chance to commit SynQuake transactions in hardware in our hybrid HyTM. We note that as with any complex parallel application, optimizing critical sections requires considerable care and effort.

We use the optimized version of SynQuake in our HyTM study henceforth. We also reduce the game physics and player action complexity to the absolute necessities. In all our experiments with SynQuake using TSX or HyTM we will use no apples, no walls, and the only action performed by players is the basic move.

Listing 4.4 shows the pseudocode for player movement in SynQuake. We note that transactional writes only occur if the player actually moves; the reason the first block of pseudocode is within the transaction is because it contains transactional reads.
for each player p1 {
    compute compound range for all possible actions
    if action is a move {
        set player speed, direction
        compute potential moving distance distance_to_move
    }
    compute action range rectangle
    collect nodes of interest from whole tree {
        overlapping_nodes = list of nodes that the
        action range rectangle would intersect
    }
    BEGIN_TXN
    for each node in overlapping_nodes list {
        for each player p2 in that area node {
            if action ranges of players p1 and p2 overlap
                if p1 and p2 can actually reach each other
                    with current speeds and distances
                        //transactional read of player p2's position
                then {
                    //collision, p1 may only move up to the collision point
                    distance_to_move = distance_between (p1, collision_point)
                }
        }
    }
    if distance_to_move != 0 {
        actually perform the player move {
            if player p1 moves across nodes {
                delete player p1 from current node //transactional write
                add player p1 to new node //transactional write
            }
            update coordinates of player p1 //transactional write
        }
    }
    END_TXN
}

The primary candidate functions to be taken out of the transaction critical section are the items 1 and 2 in Section 3.1.2. 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() calls inside the transactions. Finally, grouping several fields of certain objects for the purposes of tracking under the same version tag is performed wherever possible.
4.3.2 Initial Evaluation

This section presents an initial evaluation of the baseline TSX implementation, and highlights the need for a hybrid, multi-layered approach (such as HyTM). These results also illustrate the difficulty of getting an application to make efficient use of the hardware transactional resources via the RTM mechanism.

Figures 4.7, 4.8, and 4.9 show the percentage of transactions that commit in each layer for different versions of our TSX-based TM library, for the three quest scenarios. We show three (increasingly optimized) different versions of the two-layer TSX-only library: tsx.no_opt, tsx.opt_0, and tsx.opt_6. The results represent the average of three runs of scenarios with 500 players over 10,000 cycles on a 512 x 512 map, with an area node tree depth of 8.

![Figure 4.7: Transaction commit percentage per layer for the 1q_c scenario. For each bar, the (green) bottom layer is the pure hardware one, and the (red) top layer is the software fallback path.](image)

The three TSX versions all use a software fallback path with a single global lock, as described in [40], [77] and [78].

tsx.no_opt represents the conversion of the lock-based initial implementation of the game to a transaction-based version, and it has no further optimizations. tsx.opt_0 improves on tsx.no_opt by applying the optimizations described in Section 4.3.1 while tsx.opt_6 also adds up to 6 RTM transaction retries. Increasing the number of retries has been experimentally shown to yield diminishing returns [78, 83].

A detailed analysis of the abort causes is beyond the scope of this document, and can be found in [78]; in this section, we will merely summarize their findings.

As the figures show, with tsx.no_opt less than 3% of transactions manage to commit in the hardware layer, in any of the three quest scenarios. The main reason for this is the length of the unoptimized transactions.

Due to the reduced transaction size, tsx.opt_0 has much better commit rates in the hardware layer, up to 50% in the noq scenario. The amount of contention (determined by the specific quest scenario) is a major factor that influences the commit rates, as can be seen by comparing the percentage of...
Figure 4.8: Transaction commit percentage per layer for the \textit{4qq} scenario. For each bar, the (green) bottom layer is the pure hardware one, and the (red) top layer is the software fallback path.

Figure 4.9: Transaction commit percentage per layer for the \textit{noq} scenario. For each bar, the (green) bottom layer is the pure hardware one, and the (red) top layer is the software fallback path.
transactions that commit in the hardware layer in the 1qc scenario (with very high contention) and the noq scenario (with very low contention).

As mentioned above, a small number of transaction retries can be beneficial, and this is shown by the performance of the tsx.opt version, which achieves a commit rate of up to 95% for the noq scenario. However, yet again we notice the significant impact of contention on the commit rate: 15% commit rate in the 1qc scenario, versus almost 95% in the noq scenario.

It is worth noting that the drastically increased hardware commit rates do not always directly translate to improved overall performance [78], 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.

To conclude, the take-away point of this section is to illustrate the difficulty in getting a large and complex application to make efficient use of the hardware transactional resources via RTM. Significant optimizations and hand-tuning had to be employed, and even then, the commit rates in hardware are still not particularly impressive. This makes the case for a multi-layer, hybrid approach, to make better use of the existing transactional hardware resources.
Chapter 5

HyTM: Hybrid TM Library Design

In this Chapter 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 in [78] 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 that is described in more detail in [77, 78]. The APIs are mostly consistent with the ones described in those resources, 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 primary designer and implementor of HyTM is Mike Dai Wang [77, 78]. While I collaborated with him during the development and evaluation of HyTM, my main contribution to HyTM is extending it to support variable granularity.

In this Chapter, we first introduce the API additions that HyTM brings, including its access tracking and conflict detection mechanism, and then we present in detail the fall-back layer design of HyTM.

Finally, we show an investigation into the impact that various game parameters have on the performance of SynQuake in Section 5.3. This study indicates that among the game parameters, the main factor that influences contention is tree depth. Once this is established, we make the case that finding an ideal “sweet spot” for contention (and therefore, performance) through exhaustive search is not a feasible proposal in Section 5.4 thus further justifying the need for adaptive and dynamic variable granularity.

5.1 API Additions

HyTM’s API leverages the same set of macros as described in [77, 78] 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 tm\_types. 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 tm\_types design and Listing 5.1 provides sample code for how tm\_types are implemented. By converting objects to tm\_types, HyTM can intercept the read/write calls 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 tm\_types wrapper interface is based on the previously developed libTM library [44, 46]. A set of primitive types such as tm\_int, tm\_double, and several others are provided for programming convenience. Custom data structures and application objects can also be easily made into (tm\_types) using generics.

5.1.2 Conflict Detection with Version Tags

As described in Section 5.1.1 each shared data variable converted to 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 shows a tm\_type object containing a version tag pointer that points to a version tag; the version tag behaves similarly to a lock.
Listing 5.1: \texttt{tm\_types} wrapper interface.

```cpp
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 \rightarrow \_version\_tag);
            return (\_original\_object = new\_value);
        }
    }
}
```

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. Every time a shared data variable is written to, its version tag 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 a 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 \texttt{tm\_objects} is 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. The work and results that have been published about HyTM so far ([77] [78]) used version tag coarsening in a very limited context, and it was all done manually. In this dissertation, in Chapter 6 we describe several automated methods for selectively and adaptively changing the granularity of version tags for each individual \texttt{tm\_object}. 
5.2 Multiple-layer Fall-back Mechanism

HyTM leverages a large portion of features from the TSX-based library described in [77, 78]. In addition to the common APIs, HyTM extends the two existing transaction layers: the RTM hardware-only layer, and the last resort fall-back of locking the software global lock for the execution of the entire critical section. HyTM 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 cascading hardware transaction aborts 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 that commits are done with RTM transactions; the two software-based layers use locks to execute critical sections in order to ensure atomicity. An overview of the transaction flow path 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 [74, 77, 78], 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. 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 taken the software fall-back lock and is currently executing the critical section in one of the two software modes.
Figure 5.3: Hybrid TM transaction flow diagram.
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;
  } else {
    abort_to_layer_2();
  }

  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();
  }
```
In such case, the current RTM transaction is aborted with the \textit{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 a cascade of transactional aborts and cache thrashing (because the hardware transactions are guaranteed to abort if the software fall-back lock is unavailable). This also 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 shown in \cite{78}, 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 \cite{74,78}, incompatible instructions or system calls such as \texttt{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 \texttt{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 hardware execution of the critical section. 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 (described in more detail in Section 5.2.4.1 by not acquiring the global fall-back lock.

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 write buffer and tracks version tags associated with read shared variables into a read-set. Next, a commit phase is executed to re-validate all entries in the read-set and "flush" all entries in the write-set in one atomic operation.

#### 5.2.2.1 Compute Phase with Automatic Privatization

In the first phase of layer 2 as shown in Figure 5.3 the entire transaction is executed without using RTM or acquiring any locks. All shared reads and writes are trapped by the overloaded operators of the \texttt{tm\_types} and processed by HyTM. The access tracking strategy we implemented is Write-Buffering
Listing 5.3: HyTM layer 2: hybrid with hardware commit using RTM

```
Compute_Phase:
    // Execute critical section
    construct_read_set(); // with read version tags
    buffer_writes(); // into write-set

Commit_Phase:
    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;
    } else {
        abort_to_layer_3();
    }
    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();
    }
```
Figure 5.4: Read sets and write sets in the hardware layer and the hybrid layer with hardware commit.

(WB) with optimistic conflict detection (CD) as described in Section 2.1. 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.

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 read-set and write-set are constructed, HyTM process to the commit phase of layer 2.

### 5.2.2.2 Commit Phase

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;

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 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.

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.

We note that unless otherwise specified, in our experiments we have disabled layer 3 of HyTM (the hybrid-software layer). Our results indicate that the performance benefits that layer 3 brings are less than the overhead it causes, and therefore most of our experiments run only with layers 1, 2 and 4.

### 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 more detail in [78]. 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.
5.2.4.1 The Lemming Effect

One notable effect of the global fall-back lock used in Layer 4 is the so-called Lemming Effect \[41\]. Essentially, since the software transactions have no way to identify that hardware transactions are running concurrently, it falls upon the hardware transactions to identify running software transactions. The hardware transactions do this by reading the global fall-back lock. If a software transaction starts in Layer 4, it will acquire (and hence write) this global fall-back lock. The hardware transaction will automatically detect that an entry in its read-set was written to by another thread, and therefore the hardware transaction will abort itself. This way, a software transaction running in Layer 4 causes a sequence of cascading aborts throughout the entire system, and this phenomenon is referred to as "the Lemming Effect" \[41, 78\].

A consequence of this Lemming Effect is that whenever a transaction is running in L4, we will experience a dramatic reduction in parallelism, and hence a corresponding reduction in performance, even if transactional abort rates in the L1 and L2 layers are low.

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 hash tables 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.

5.2.6 Optimized HyTM Results

As noted in Section 4.3.2, the bar called hytm in the graph uses the unoptimized transactions similar to tsx.no_opt and relies on HyTM for automatic optimizations. Additional manual optimizations such as application-level code changes are not performed by HyTM. In this section, we show results of hytm.opt, which contains manually reduced transaction sizes in SynQuake itself in addition to using HyTM as discussed in Section 4.3.1.

Figures 5.5, 5.6, and 5.7 show the transaction composition for all the tsx versions with the addition of hytm and hytm.opt, for a simulation with 500 players, 10,000 game cycles, and on a 512 x 512 game map. Under all game quest scenarios, hytm.opt showed a significant increase in the fraction of transactions committed in the HW.Only layer. In the low contention quest scenario, hytm.opt had the greatest impact on the HW.Only layer, resulting in an increase from 26% to 90% when compared to hytm. The reason for a higher commit rate for the hardware transactions for hytm.opt is the presence of multiple layers, which prevents transactions that fail in Layer 1 from falling back directly to Layer
Figure 5.5: hytm.opt transaction composition for the 1qc quest scenario.

Figure 5.6: hytm.opt transaction composition for the 4qq quest scenario.
4, which would in turn abort all other subsequent hardware transactions. In other words, the multiple layers alleviate the Lemming Effect described in Section 5.2.4.1.

Throughout the remainder of this work, we will be using hytm.opt in all our experiments.

5.3 Sensitivity of SynQuake to Varying Game Parameters with HyTM

Similar to the sensitivity study that we conducted for SynQuake running with libTM in Chapter 4 in this section, we investigate the impact of several game parameters on contention in SynQuake with HyTM. To this end, we varied the player speed, map size, and tree depth, for all three quest scenarios (1 central quest 1qc, one quest in each quadrant 4qq, and no quests noq), for both fixed granularities: entity (fg.ent) and set (fg.set).
Figure 5.8: Execution time with various game parameters for the 1\textit{qc} scenario. The X axis parameters are, from top to bottom: tree depth, player speed and map size.

Figure 5.9: Execution time with various game parameters for the 4\textit{qq} scenario. The X axis parameters are, from top to bottom: tree depth, player speed and map size.
Figure 5.10: Execution time with various game parameters for the noq scenario. The X axis parameters are, from top to bottom: tree depth, player speed and map size.

Figure 5.11: Transaction commit percentages per layer for layers 1, 2 and 4 (from bottom to top), for the lqc scenario. In each bar group, the first bar is for ent granularity, and the second bar is for set. The X axis parameters are, from top to bottom: tree depth, player speed and map size.
Figure 5.12: Transaction commit percentages per layer for layers 1, 2 and 4 (from bottom to top), for the 4qq scenario. In each bar group, the first bar is for ent granularity, and the second bar is for set. The X axis parameters are, from top to bottom: tree depth, player speed and map size.

Figure 5.13: Transaction commit percentages per layer for layers 1, 2 and 4 (from bottom to top), for the noq scenario. In each bar group, the first bar is for ent granularity, and the second bar is for set. The X axis parameters are, from top to bottom: tree depth, player speed and map size.
Figures 5.8, 5.9 and 5.10 show the execution times of both fixed granularities for all three quest scenarios, for all combinations of the three varying game parameters.

Figures 5.11, 5.12 and 5.13 show the percentage of transactions that committed in each of the three layers (1, 2 or 4), for each of the three quest scenarios respectively.

In all figures, the first bar in each group of 2 bars represents the execution for fixed granularity $ent$, and the second bar represents the execution for fixed granularity $set$. Similarly, for all six figures, the $X$ axis contains the specific combination of game parameters for that particular run; specifically, these parameters are, from top to bottom: tree depth, player speed, and map size.

The main observation from these figures is that of the three game parameters, the tree depth has the most significant impact on both execution performance, and the amount of transactions that commit successfully in the hardware layers.

This can be observed in the execution time figures by noticing that there is a sharp decrease in execution time, in each group of three consecutive bar clusters (i.e., varying the tree depth, while maintaining the other two parameters constant). This is not the case when looking at every third bar group (i.e., varying player speed) or every ninth bar group (varying map size).

Likewise, the figures showing the commit percentages of transactions exhibit a similar pattern: the number of transactions that commit in the hardware layers increases drastically as the tree depth rises from 4 to 8, but does not change significantly as we vary the other two game parameters.

This can be explained by the fact that the tree depth is the factor that most closely models contention of all three game parameters, and contention is the main factor that influences the execution time, and the layer in which transactions are more likely to commit.

An interesting case shows for the 4qq scenario: the tree depth only matters for the smallest map size (256), but no longer has any impact once we move to larger map sizes. This is because for the smallest map size, even though players gather around quests situated in the center of each quadrant, the small size of the map means that players from one cluster may potentially interact with players from another cluster. This is due to the way that the game computes objects with which a player might potentially interact, based on the player’s speed and the distance between the objects. If the distance between the player and another object is small enough, then the object is considered "in range" for that player, and a potential interaction between the player and the object is computed. Once the map gets large enough, the player clusters are sufficiently far from each other that interactions are no longer possible between players from different clusters, which effectively eliminates contention, and therefore varying the tree depth (which models contention) or any other parameters has a negligible impact.

5.4 Looking for the Sweet Spot

Since the tree depth is the parameter that best models contention, managing contention by adjusting the tree depth became an obvious path to follow. We examined the impact of various fixed tree depths on the three quest scenarios, for 4000 players, over 4000 cycles, on a map size of 512 x 512.
Figure 5.14: Execution time for the 1qc quest scenario, for various fixed tree depths (shown on the X axis). In each bar group, the first bar represents entity granularity, and the second bar is set granularity.

Figure 5.15: Execution time for the 4qq quest scenario, for various fixed tree depths (shown on the X axis). In each bar group, the first bar represents entity granularity, and the second bar is set granularity.
Chapter 5. HyTM: Hybrid TM Library Design

Figure 5.16: Execution time for the noq quest scenario, for various fixed tree depths (shown on the X axis). In each bar group, the first bar represents entity granularity, and the second bar is set granularity.

Figure 5.17: Transaction commit percentage per layer for layers 1, 2 and 4, for the 1qc scenario. In each bar group, the first bar represents entity granularity, and the second bar is set granularity. For each bar, layer 1 is the bottom-most one, and layer 4 is the top one.
Chapter 5. HyTM: Hybrid TM Library Design

Figure 5.18: Transaction commit percentage per layer for layers 1, 2 and 4, for the 4qq scenario. In each bar group, the first bar represents entity granularity, and the second bar is set granularity. For each bar, layer 1 is the bottom-most one, and layer 4 is the top one.

Figure 5.19: Transaction commit percentage per layer for layers 1, 2 and 4, for the noq scenario. In each bar group, the first bar represents entity granularity, and the second bar is set granularity. For each bar, layer 1 is the bottom-most one, and layer 4 is the top one.
Figures 5.14, 5.15 and 5.16 show the execution times for the three quest scenarios. Figures 5.17, 5.18 and 5.19 show the transaction commit percentage per layer for the same quest scenarios.

In each bar group, the first bar represents entity granularity, and the second bar is set granularity.

From the first three figures, we note that the ideal tree depth seems to be situated around a tree depth of 8 or 9 for all three quest scenarios. A natural question would then be: why do we need any heuristic at all? We can just run once through all reasonable tree depths, find this sweet spot, and always use that from that point onward.

However, this is not the case.

Figures 5.20, 5.21 and 5.22 show that the sweet spot changes significantly depending on the number of players, even for the same, and simplest, quest scenario (noq).

**Execution time for noq 250 players**

![Execution Time Chart](image)

Figure 5.20: Execution time with 250 players for the noq quest scenario, for various fixed tree depths (shown on the X axis). In each bar group, the first bar represents entity granularity, and the second bar is set granularity.

We can see that the sweet spot for 8000 players seems to be somewhere around a tree depth of 8 to 10, for 16000 players it is somewhere between 8 and 14, and for 250 players it is less than 8. Therefore, a "universal" sweet spot doesn’t exist, since it is likely that different machines will run with different numbers of players, different quest scenarios, and so on.

Moreover, it is worth noting that running an exhaustive profiling operation trying to determine an "optimal" configuration for a given set of parameters, is likely to be infeasible and very time-consuming (on the order of days), and may need to be re-done every time there are any significant changes in the hardware, software, or some of the game parameters that were not covered previously.
5.5 Performance Comparison between HyTM SynQuake and Lock-Based SynQuake

In this Chapter, we compared the performance of SynQuake running under different versions of our HyTM or TSX-based libraries. In this Section, we compare the performance of HyTM SynQuake against a lock-based implementation of SynQuake.

While both the transactional SynQuake version and the lock-based version of SynQuake that we use are similar to the existing respective versions of SynQuake developed as part of the initial benchmarking effort [46], the codebases are not identical. As we mentioned, we had to reduce the size of the SynQuake transactions significantly due to hardware constraints in TSX.

Second, merely comparing execution times of lock-based and transaction-based implementations does not provide an accurate picture, since it does not quantify or include the effort required to achieve an efficient implementation of fine-grained locks, an effort that is higher than parallelizing a piece of code using transactional memory.

With these caveats, we present the comparison of execution times of HyTM against the lock-based SynQuake implementation, as this is the closest and most meaningful related work that we can use as a baseline for comparison.

In all cases, the experiments were run with 4000 players, for 4000 cycles, and on a map of size 512 x 512. All other game configuration parameters were kept the same between the two versions of SynQuake in order to provide a comparison that is as fair and meaningful as possible.

Figures 5.23, 5.24 and 5.25 show the execution times of two versions of lock-based SynQuake and HyTM SynQuake running with several fixed tree depths. We are using the same fixed tree depths that we use for evaluating our heuristics in the context of HyTM in Chapter 6. The lock-based code has two variants: lock-tree and lock-leaves. The first one is similar to a coarse-grain lock implementation, while the latter is closer to a more efficient fine-grained implementation that tries to minimize the amount of tree area nodes that are locked, as well as the duration of their locking.

For HyTM, the two bars in the graph represent starting granularities of ent and set, respectively.

The original publication comparing libTM SynQuake with locks showed that libTM SynQuake outperformed locks. In contrast, we can see from our experiments that the lock-based code outperforms the HyTM code in all cases. The reasons for this lie in the reduced complexity of game physics and player actions that we use in our game.

In order to achieve shorter transaction duration that has a chance to commit in hardware, in our experiments with SynQuake we use no walls and player actions are reduced to player moves. In contrast, in the original SynQuake work, more game physics i.e., collisions with walls, and complex actions e.g., move and eat, or move and fight were used. Because lock holding needs to be conservative for all objects overlapped by the bounding box of the player from the beginning to the end of a complex action in the lock-based SynQuake, complex actions induced more false sharing in the original lock-based SynQuake versus TM version of SynQuake. Moreover physics computation would prolong the duration of lock holding while not creating any conflicts for the TM SynQuake.

With our shorter transactions, we have fewer opportunities to hide TM overheads with HyTM. On the upside, having a simpler game makes it possible to analyze and characterize the sources of contention in the game, as we show in Chapter 6.
Chapter 5. HyTM: Hybrid TM Library Design

Figure 5.21: Execution time with 8000 players for the noq quest scenario, for various fixed tree depths (shown on the X axis). In each bar group, the first bar represents entity granularity, and the second bar is set granularity.

Figure 5.22: Execution time with 16000 players for the noq quest scenario, for various fixed tree depths (shown on the X axis). In each bar group, the first bar represents entity granularity, and the second bar is set granularity.
Lock-based SynQuake vs HyTM SynQuake for 1qc

Figure 5.23: Execution time of lock-based SynQuake vs HyTM SynQuake for the 1qc scenario. For the lock-based versions, the two bars in each group represent fine-grained locking (leaves) and coarse-grained locking (tree), respectively. For HyTM, the two bars represent starting ent and set granularities, respectively.

Lock-based SynQuake vs HyTM SynQuake for 4qq

Figure 5.24: Execution time of lock-based SynQuake vs HyTM SynQuake for the 4qq scenario. For the lock-based versions, the two bars in each group represent fine-grained locking (leaves) and coarse-grained locking (tree), respectively. For HyTM, the two bars represent starting ent and set granularities, respectively.
Lock-based SynQuake vs HyTM SynQuake for noq

Figure 5.25: Execution time of lock-based SynQuake vs HyTM SynQuake for the noq scenario. For the lock-based versions, the two bars in each group represent fine-grained locking (leaves) and coarse-grained locking (tree), respectively. For HyTM, the two bars represent starting ent and set granularities, respectively.
Chapter 6

Variable Granularity in HyTM

In this Chapter, we first present the mechanism by which we can vary granularity in HyTM and its applicability to SynQuake in Section 6.1. Then, in the same Section, with the help of HyTM statistics, we present a detailed investigation into the sources of contention in HyTM and their relative importance in influencing performance. Next, we introduce several heuristics for varying granularity in Sections 6.3 and 6.4. We present an evaluation of these heuristics, and investigate the effects of combining several heuristics in Section 6.5.

6.1 Mechanism for Varying the Tracking Granularity in HyTM

For the purposes of our prototype implementation of variable granularity in HyTM, we add a function called assign_tag() to the standard TM interface of HyTM. This function creates a mapping between an application object and any pre-existing HyTM-maintained statistical object, called the object tag, as shown in Figure 6.1. The same function allows the application to manipulate this mapping for any of its objects.

Figure 6.1: A regular object is transformed into a tm_object by encapsulating metadata and a version tag pointer (a pointer to a version tag object) together with the original object. The tag object contains the tag version number and a conflict counter.

The object tag maintains HyTM statistics for the respective application object. Each object tag currently contains two fields: a version number, and a conflict counter. The version number of the
Chapter 6. Variable Granularity in HyTM

An object is incremented during the commit of an update transaction which has performed one or more transactional writes on the respective object. During read-validation, the recorded value of the object’s tag version number is compared against the current value of the object’s tag version number. If the value of the tag’s version number has changed from a previous read, the transaction will abort. The conflict counter is incremented every time that a read validation check on this object fails.

The assign_tag() HyTM API function can be used by the application for assigning, and reassigning tags to its objects. Accessing the per-object conflict counters is currently performed through direct accesses, but it is trivial to augment the library with a read_counters() function that returns the specific values for a tm_object.

Varying the tracking granularity of an application object can be done by simply reassigning its tag pointer to point to a different object tag. More specifically, coarsening the tracking granularity on the fly is equivalent to assigning a single object tag multiple tm_objects. Figure 6.2 shows such a coarse granularity scenario, where multiple tm_type objects are sharing the same object tag. All objects sharing the same TM-maintained object tag will be tracked as a unit for the purposes of conflict detection by the TM.

If false sharing is not a problem, then managing a smaller number of version tag objects may gain significant performance improvements in HyTM because of better cache behaviour. The cache behavior is expected to improve with a lower number of (disjoint) object tag accesses and fewer cache replacements of object tags from the cache.

Aggregating multiple objects to use the same version tag has an additional benefit in the case of HyTM: for HyTM with coarser granularities, the reduction in the number of tags that need to be checked at the read validation stage leads to a shorter read validation phase, hence a shorter commit phase, and therefore a higher chance of successful hardware commit.

Since, as we have shown, one of the major causes of hardware transactional aborts is the (hardware) transaction size, and the read set of the software transaction to be committed in hardware is usually much larger than the write set, coarse granularity tracking may have significant performance impact.

Besides providing for an indirection point which allows varying the conflict tracking granularity at run-time, the reason for keeping track of object statistics outside the tm_object is as follows: if the statistic counters were part of the tm_object, then increments to any of the counter fields would, in effect, modify the tm_object itself thus causing aborts to hardware transactions that were just reading the respective object, as would be the case with the TSX implementation of hardware TM. By keeping the tag version number and any counters used for statistic purposes separate from the tm_object itself, we avoid this scenario. Any thread can update statistics on the objects it uses without causing unnecessary aborts of hardware transactions reading those same objects.

6.1.1 Tag Granularities Used in SynQuake

Each area node has a hashtable containing all the players that are currently in the map region covered by that area node, as explained in Section 2.3.2 and as shown in Figure 2.11.

For SynQuake, we use two levels of tracking granularity: fine (object-level) granularity and coarse granularity where each object tag covers multiple tm_objects i.e., players. When we expect a high degree of contention among a subset of the shared objects in the game (i.e., between players), we switch the object tag pointers of these players to point to their own individual object tag. This is the fine granularity we mentioned above, and we will refer to it from now on as entity-level granularity. When we expect a
Figure 6.2: Multiple tm_type wrapped application objects sharing the same version tag, which acts similar to a lock. Grouping transactional objects to use the same version tag also reduces the size of the read set of transactions, making them more likely to commit in hardware.

relatively low degree of contention among a subset of players in an area corresponding to a specific area node, we set the object tag pointers of those players to point to the single object tag associated to the area node that those players belong to. We will refer to this granularity as set-level granularity, as each individual tag covers a set of objects.

Figure 6.3 illustrates this scenario of coarse granularity tag objects in SynQuake, where multiple players share the tag object of their area node.

Note that the granularity of each individual object may be entity or set, independent of the granularity of any other objects; thus, it is entirely possible for some players belonging to an area node to have coarse granularity, while at the same time other players, owned by the same thread and inside the same area node, to have fine granularity at the same time.
6.1.2 Detailed Investigation of Contention Sources based on HyTM-provided statistics

In this Section, we provide additional insight into the behaviour of HyTM and SynQuake based on the information collected through conflict counters in HyTM. All experiments were run with 4000 players and over 4000 cycles (leading to a total of 16 million transactions), and the static load balancing where we assign one quadrant of the game map to one of 4 threads as described in Section 2.3.4. All results represent the average over three runs. The values for the conflict counters are collected in the last cycle of the simulation.

Figures 6.4, 6.5 and 6.6 show the execution times for HyTM SynQuake for all three quest scenarios, for two fixed tree depths (4 and 14), for entity and set granularity, and for three map sizes: 1024x1024, 512x512, and 256x256. We chose the tree depths as the two extreme values that might be reasonably used for the chosen map sizes.

Figures 6.4, 6.5 and 6.6 also show the transaction commit percentage per layer for layers 1, 2 and 4 (from bottom to top), for entity and set granularity, for all three quest scenarios, for the same two fixed tree depths, 4 and 14, and for the same three map sizes: 1024x1024, 512x512 and 256x256.

Tables 6.1, 6.2 and 6.3 show selected values for the conflict counters for three map sizes: 1024x1024, 512x512, and 256x256 respectively. Once again we show two tree depths, 4 and 14, and we show both entity and set granularities. The values in the tables represent the sum of the conflict counters for the respective objects they are associated with, for the duration of the run. For entity we show conflict counters for players (cc\_player) and for the area node hashtables (cc\_hash), while for set granularity we show conflict counters for the area node (cc\_areanode) and for the area node hashtable. We don’t show the conflict counters for area nodes for entity granularity, or the conflict counters for players for set granularity, because they are zero: for entity granularity, player version tag pointers point to the player’s own tag object, and not the area node’s tag object, and therefore the area node’s tag object is not being used; similarly, for set granularity, player version tag pointers point to the tag object of the area node the player is in, and the player’s own tag object is not being used.

Figure 6.3: On the left, a SynQuake player has its own version tag object. On the right, multiple SynQuake players (P2 and P3) share a common (coarse-grain) version tag object: the one of the area node corresponding to the map region where all these players are.
Figure 6.4: Execution times and transaction commit rates per layer for 2 fixed tree depths, 4 and 14, and all quest scenarios for a map size of 1024x1024. In the right figure, layers are shown from layer 1 (the hardware layer) at the bottom, to layer 4 (the software layer) at the top.

Figure 6.5: Execution times and transaction commit rates per layer for 2 fixed tree depths, 4 and 14, and all quest scenarios for a map size of 512x512. In the right figure, layers are shown from layer 1 (the hardware layer) at the bottom, to layer 4 (the software layer) at the top.

Figure 6.6: Execution times and transaction commit rates per layer for 2 fixed tree depths, 4 and 14, and all quest scenarios for a map size of 256x256. In the right figure, layers are shown from layer 1 (the hardware layer) at the bottom, to layer 4 (the software layer) at the top.
Table 6.1: Conflict counter values for players, area nodes and hash tables for all three quest scenarios, for entity and set granularities, and for two tree depth levels, 4 and 14, for a map of size 1024x1024. We do not show the conflict counters for players for set granularity, or the conflict counters for area nodes for entity granularity, since they are zero. Values represent averages over three runs.

<table>
<thead>
<tr>
<th></th>
<th>ent_4</th>
<th>set_4</th>
<th>ent_14</th>
<th>set_14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cc_player</td>
<td>cc_hash</td>
<td>cc_areanode</td>
<td>cc_hash</td>
</tr>
<tr>
<td>1qc</td>
<td>178.33</td>
<td>15085</td>
<td>270105</td>
<td>14160</td>
</tr>
<tr>
<td>4qq</td>
<td>23.66</td>
<td>1793</td>
<td>14772</td>
<td>1688</td>
</tr>
<tr>
<td>noq</td>
<td>145.67</td>
<td>46405</td>
<td>214721</td>
<td>45999</td>
</tr>
</tbody>
</table>

The conflict counters are incremented when their associated object causes a read validation to fail, as described in Section 6.4.2.

Table 6.2: Conflict counter values for players, area nodes and hash tables for all three quest scenarios, for entity and set granularities, and for two tree depth levels, 4 and 14, for a map of size 512x512. We do not show the conflict counters for players for set granularity, or the conflict counters for area nodes for entity granularity, since they are zero. Values represent averages over three runs.

<table>
<thead>
<tr>
<th></th>
<th>ent_4</th>
<th>set_4</th>
<th>ent_14</th>
<th>set_14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cc_player</td>
<td>cc_hash</td>
<td>cc_areanode</td>
<td>cc_hash</td>
</tr>
<tr>
<td>1qc</td>
<td>451</td>
<td>66180</td>
<td>409514</td>
<td>60941</td>
</tr>
<tr>
<td>4qq</td>
<td>21</td>
<td>1320</td>
<td>8945</td>
<td>1190</td>
</tr>
<tr>
<td>noq</td>
<td>975</td>
<td>50149</td>
<td>487286</td>
<td>44295</td>
</tr>
</tbody>
</table>

Table 6.3: Conflict counter values for players, area nodes and hash tables for all three quest scenarios, for entity and set granularities, and for two tree depth levels, 4 and 14, for a map of size 256x256. We do not show the conflict counters for players for set granularity, or the conflict counters for area nodes for entity granularity, since they are zero. Values represent averages over three runs.

<table>
<thead>
<tr>
<th></th>
<th>ent_4</th>
<th>set_4</th>
<th>ent_14</th>
<th>set_14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cc_player</td>
<td>cc_hash</td>
<td>cc_areanode</td>
<td>cc_hash</td>
</tr>
<tr>
<td>1qc</td>
<td>98</td>
<td>4518</td>
<td>67593</td>
<td>4739</td>
</tr>
<tr>
<td>4qq</td>
<td>21</td>
<td>1320</td>
<td>8945</td>
<td>1190</td>
</tr>
<tr>
<td>noq</td>
<td>975</td>
<td>50149</td>
<td>487286</td>
<td>44295</td>
</tr>
</tbody>
</table>

Not shown in any of the tables are the following numbers, which we will also use in our analysis. The number of writes due to player moves across area nodes, and their associated hash tables for the configuration with the shallow tree are: 64-68K for insertion (and also for deletion) in 1qc, 160K for 4qq, and 1 million for noq, for the large map. The same counts for hash tables insertions/deletions for the shallow tree configuration are 58-62K for 1qc, 149-153K for 4qq and 1.4 million for noq in the medium map. For the small map with the shallow tree, they are 85-95K for 1qc, 109-123K for 4qq and 1 million for noq for the small map. The numbers in the ranges reported apply to both entity and set configurations on each map.

Table 6.4 shows the number of area nodes (in millions) visited while building the list of area nodes of interest for each player in SynQuake transactions, as described in Listing 3.1 in Section 3.1.2. We show these values for all three quest scenarios, for both entity and set granularity, for two tree depths, 4 and 14, and for three map sizes: 1024x1024, 512x512, and 256x256. The area nodes of interest are the area nodes with which a player’s bounding box overlaps, and during the transaction the player may
Table 6.4: Number of area nodes (in millions) visited while building the list of nodes of interest in SynQuake transactions. For each player, the nodes of interest are the nodes that cover areas of the map which are overlapping with the player’s bounding box, which is the range that the player may cover in the current timestep. We show data for three map sizes: 1024x1024, 512x512, and 256x256. We also show data for two tree depths, 4 and 14, and for both entity and set granularity.

<table>
<thead>
<tr>
<th></th>
<th>1024x1024</th>
<th></th>
<th>512x512</th>
<th></th>
<th>256x256</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>depth_4</td>
<td>depth_14</td>
<td>depth_4</td>
<td>depth_14</td>
<td>depth_4</td>
<td>depth_14</td>
</tr>
<tr>
<td>ent set</td>
<td></td>
<td></td>
<td>ent set</td>
<td></td>
<td>ent set</td>
<td></td>
</tr>
<tr>
<td>1qc</td>
<td>166</td>
<td>166</td>
<td>166</td>
<td>165</td>
<td>165</td>
<td>166</td>
</tr>
<tr>
<td>4qq</td>
<td>151</td>
<td>150</td>
<td>151</td>
<td>150</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>noq</td>
<td>146</td>
<td>146</td>
<td>148</td>
<td>148</td>
<td>148</td>
<td>148</td>
</tr>
</tbody>
</table>

potentially interact with any player in one of those area nodes.

Looking at these graphs, and the associated commit break-downs and counters, we notice several trends:

- The tree depth setting makes more impact than the entity/set parameter setting.
- The 1qc quest scenario has the longest execution times for the shallow tree, for all map sizes; the execution time of this quest scenario benefits from the deep tree in all map sizes.
- The 4qq quest scenario has the best execution time out of all three quests, for each tree configuration, respectively, for all map sizes.
- The execution times are better for the deep tree versus the shallow tree for all quest scenarios in the large map size. This trend is mostly reversed for the small map size.

We explain these trends by investigating the sources of contention and explaining how they act in the three game scenarios and 3 game map configurations, in detail.

In the 1qc scenario, most players are very close to each other, causing many collisions upon player moves. Due to this, the number of moves with non-zero distance that produce a transactional update of a player’s position is low, which lowers the conflict rate on player data for 1qc. On the other hand, only a few hash tables host most of the players in the game in 1qc; the high density of players in these hash tables, coupled with their location at the boundary between thread allocations determines a high number of hash table conflicts compared to the data conflicts (15K versus 178 on the large map). The reason for this is that in our L2 layer of HyTM, the tags on hash tables are coarse, one tag per hash table, and each hash table of an area node contains all players situated within the area node. Therefore any player move across area node hash tables may invalidate all concurrent hash table reader transactions on both origin and destination hash tables. Section 3.3 shows the data structure for hashtables in libTM; the hashtables in HyTM have the same data structure, except for the lock being replaced by the version tag. In the same Section 3.3 we also explain how hashtables are a significant source of contention and represent a bottleneck, and the cause of a large number of true conflicts. The explanations in Section 3.3 are for libTM, but they apply to HyTM as well, as the two libraries share similar data structures, and experience similar access patterns and experimental trends.

We see that, for 1qc with the deep tree, where each hash table of an area node covers much less of the game map, hence fewer players, the conflict counts on the hash tables are significantly decreased, e.g., from 15K for the shallow tree, to 2.9K for the 1qc with deep tree for the large map. Although the
tracking granularity in hardware is much finer than the L2 tags, we see that the commits in hardware spike between 12% for the shallow tree to 70% for the deep tree on the same (large) map for the 1qc scenario; this points to the possibility of a similar bottleneck in the L1 hardware layer as well.

We see a similar effect for 4qq with deep tree for the large map, where the hash table conflict counts are reduced, and the hardware commits are increased when we switch from the shallow to the deep tree configuration.

Finally, due to the extreme degree of player packing in 1qc, a record high number of transactional reads occurs in the software HyTM transaction, where for each player, we read the positions of all neighbor players, i.e., players in area nodes overlapping the player’s action range. This has several side effects.

First, the transaction size and the computation that is lost on abort is much higher than with other scenarios with fewer neighbors per player; therefore, for each abort due to hash table conflicts in 1qc, significant computation on reading neighbor players for collision detection is lost, and needs to be redone. Second, the cache footprint of the transaction as a whole is high, due to both the transactional reads of neighbor player positions themselves, and the writes done by HyTM for bookkeeping of these transactional reads. The high cache footprint increases the probability of hardware abort of all hardware transactions, due to cache replacement or invalidation. The combination of long transactions, with non-negligible abort rates and large cache footprint and or invalidations leads to low performance for the shallow tree case with 1qc. The deep tree helps reduce the contention on the few hash tables at the boundary, hence alleviate the conflicts on hash tables, because player moves are spread across more of these. Although conflicts are tracked at a finer granularity in hardware, reducing hash table contention helps increase the fraction of hardware commits and performance in all maps.

We see similar effects for 4qq for the large and medium maps for increasing the tree depth. Here as well, the hash table conflict counts are reduced, and the hardware commits are increased for the large and medium maps, from the shallow to the deep tree configuration. This trend is reversed, however, in the small map size, where only for the 1qc quest scenario the deep tree improves performance. Even in the 1qc quest scenario, the performance improvement with increasing the tree depth is much smaller for the small map size compared to the large map size. The reason for this is as follows: the number of hash tables can be high as long as players don’t move much or the player density is low. As shown for the 1qc scenario, contention on hash tables in area nodes located at the boundary between thread allocations is of particular performance concern. Due to the location of the quests in the centers of each thread’s allocation, contention on boundary hash tables is much less a problem in 4qq. On the other hand, a high number of hash tables overall hurts performance if players move frequently, creating many writes on potentially many hashtable objects that need to be committed in hardware. Although player movement is still somewhat restricted by quests in 4qq, players move more than in 1qc where player crowding is the worst. Player movement is most frequent in the noq scenario where the effect of a large number of hash tables coupled with frequent player moves is shown most clearly in the deep tree case.

Performance for noq is better than for 1qc and comparable to 4qq, on each map size taken separately, because each player has the fewest neighbors in noq versus 4qq and 1qc, hence transaction sizes are shortest. However, most player moves occur in noq, because there are fewer player collisions than in the other quest scenarios, leading to worse performance than 4qq.

It is likely that the high number of writes due to player moves with the shallow tree (2x1 million for noq, versus 2x123K for 4qq, versus 2x90K for 1qc) exceeds the hardware capacity even when they occur
on relatively few hash tables, as in the case of the large and medium maps with noq with the shallow tree. As the number of area nodes and associated hash tables increases with the depth of the tree, these numbers are correspondingly higher, leading to the highest overall abort rate for noq with the small map and the deep tree. As we can see, slightly over 50% of all transactions abort in both L1 and L2 and execute sequentially in the L4 software fall-back path.

Regardless of quest scenario, smaller maps generate more read activity than larger maps because we use the same number of players and the same player speed for all maps. Hence, player density increases on smaller maps where each player has more neighbors in its action bounding box, hence read set. Finally, more area nodes are processed when detecting overlap of player action bounding boxes at the same player speed for smaller maps because the bounding box is larger relative to the grid unit. These effects are most obvious for the small map with noq and the deep tree. A large number of reads to area nodes (1.2 billion) is added to the already large number of writes due to moves across hash tables (2x1 million) in this case; as we can see, as a result, the number of software fall-back transactions is the highest in this configuration.

It is interesting to note that even for the noq configuration, fine grain splitting the game map in the deep tree configuration reduces the conflicts on hash tables significantly. However, the increase in the total number of hash table tags offsets the gains from conflict reduction.

As we can see from the tables, the number of conflicts on hash tables is highest in the noq scenario for almost all scenarios. One of the reasons for this is that the number of commits in L1 hardware is lowest in noq; in the other quest scenarios we simply don’t get to increment conflict counts when the transaction commits directly in L1. The other reason for this was already mentioned above and we reiterate it here. In the 4qq and noq scenarios, players move a non-zero distance more frequently compared to 1qc. This includes moving more frequently across area nodes and their associated hash tables. In the 4qq scenario, however, due to the location of the quests, players rarely interact or move across thread allocation boundaries. This is shown by the low number of conflicts on both players and hash tables in 4qq for all configurations. In contrast to 4qq, in the noq scenario, the frequent collision free moves in noq result more frequently in moving across area nodes and also between hash tables allocated to different threads.

Finally, we notice that, in all cases where a significant number of hash table conflicts is present, performance and the commit rates in layers L1 (hardware) and L2 of our Hybrid TM are relatively insensitive to the number of data conflicts. Specifically, the performance and commit rate breakdowns are very similar for the entity and set configurations of 1qc with the shallow tree in all three map sizes, even when the data conflicts spike from 178 to 270K for the entity versus area node conflicts in the large map.

In summary, the main underlying contention bottleneck is due to contention on boundary hash tables and/or large numbers of player moves across hash tables, regardless of location.

In the above, boundary hash tables are hash tables associated with area nodes located close to thread allocation boundaries; these hash tables become contended when the player density in their area is high, as in 1qc. Conversely, a large number of hash tables create a bottleneck due to significantly increasing the total number of writes in a transaction, irrespective of hash table location, and even when the per-hash table player density is relatively low. This happens when the player moves are unobstructed by quests, hence frequent, as in noq.
6.2 Heuristics for Varying the Tracking Granularity in HyTM

Our intuition is that we can vary the shared object tracking granularity dynamically, on the fly, in a speculative parallelization framework, with the purpose of maintaining a balance between avoiding false sharing, by refining the granularity on one hand, while reducing the tracking overhead, by keeping track of as few objects as possible, on the other hand.

In this Chapter, we show that it is possible to apply this idea in the context of our HyTM system and our SynQuake realistic game application and obtain significant performance improvements.

Towards this, we leverage the mapping that already exists in HyTM between an application object and an associated HyTM-maintained object, called the object tag. We expose this mapping to the application as part of the HyTM interface. This allows the application to manipulate this mapping for its objects, which enables changing the HyTM conflict tracking granularity for application objects at run-time. Exposing the mapping also allows the application to gain access to per-object statistics maintained by HyTM, particularly the contention level of each application object.

We introduce heuristics for varying the HyTM tracking granularity. These heuristics use and update the granularity for two categories of data structures on which contention may occur: metadata objects (hashtable tags) and application objects (player tags). As explained in Section 2.3.2 and as shown in Figure 2.1b, the metadata objects consist in one hashtable per each area node which contains all the players that are currently in the map region covered by that area node. As such, we classify our heuristics in two categories, which we will from now on refer to as metadata heuristics, and data heuristics.

The application can use either its own information or statistics provided by HyTM in order to decide when changing the tracking granularity on either data or meta-data is appropriate. Application-specific knowledge such as player density, proximity to regions owned by other threads can help predict conflicts and change TM tracking granularity proactively, before conflicts occur. Alternatively, the application can use heuristics based on the per-object statistics that the HyTM library collects and provides as part of the object tag mappings in order to reactively change TM tracking granularity for its objects.

Our heuristics assign a finer tracking granularity for sets of game objects that either experience or are predicted to experience a lot of conflicts due to concurrent accesses by different threads. The finest granularity can be obtained by tracking each object individually. Coarser granularities can be obtained by associating objects colocated within a map partition to the same HyTM conflict tracking unit.

The following sections present our heuristics for managing contention through dynamic restructuring of the tree meta-data as well as finer-grained data heuristics for variable granularity at the object level.

Section 6.3 describes our fine-grained, object-level data heuristics. Section 6.4 presents our metadata heuristics for adjusting tree depth in order to manage contention on metadata structures. We present both a heuristic based on application-specific knowledge in Section 6.4.4 and a heuristic that relies only on data provided by the underlying TM library in Section 6.4.2. We also discuss the effect of combining metadata heuristics and data heuristics together in Section 6.5.

We note that unless otherwise specified, in our experiments we have disabled layer 3 of HyTM (the hybrid-software layer). Our results indicate that the performance benefits that layer 3 brings are less than the overhead it causes, and therefore most of our experiments run only with layers 1, 2 and 4.
6.3 Data Heuristic: Proactive Conflict-Predictive Adaptation (CP)

This Section presents our data heuristic for variable granularity tracking at the game object e.g., player object level.

We use a Conflict-Predictive algorithm which proactively predicts conflicts, and changes the tracking granularity of game objects before it may become necessary to do so. Specifically, by default, we use coarse (set) granularity in HyTM for all game objects. However, when a player approaches the boundary region with the game region assigned to a different thread, we proactively switch the tracking granularity for that player to fine (entity).

As we explain in Section 6.1.1, players tracked at the entity granularity use their own individual tag objects, while players tracked at the set granularity use the tag object of a specific area node, e.g., the area node that they belong to.

![Figure 6.7: Performance in HyTM of the conflict-predictive (CP) adaptation for the three quest scenarios. The three bars in each group represent, respectively, the execution time for fixed entity granularity, fixed set granularity and the conflict-predictive adaptation.](image)

Figure 6.7 shows the performance of our heuristic, relative to the two baselines of fixed entity granularity and fixed set granularity. The heuristic has comparable performance with the baselines for the 4qq and noq scenarios, but does slightly worse than fixed set granularity for the 1qc scenario. Overall, the heuristic does not significantly improve performance. However, with this heuristic, we have attempted to improve data contention while a significant source of contention still remains due to shared meta-data access in the game. As we will see in the following sections, for cases of data contention, it is only when meta-data contention on hash tables that maintain players is also addressed that any performance improvement can be achieved.
6.4 Metadata Heuristics

This section presents our meta-data heuristics for variable granularity tracking of in-game meta-data. These heuristics dynamically adjust the structure of the Area Node Tree that maintains the game objects throughout the duration of the program. Because they affect the area node tree, hence the allocation of players to area nodes, the meta-data heuristics can affect contention on both game meta-data and game data.

Since the location of the shared objects in the tree for spatial search data structure maintained by SynQuake corresponds to their location within an application world or map, we can selectively increase the number of map splits, hence the depth of the subtrees in which highly contended objects are stored. Conversely, we can reduce the depth of the subtrees that hold objects that are infrequently accessed concurrently.

We call our metadata heuristic selective node splitting and we implement two versions of it. In the first version of node splitting, we use only HyTM provided statistics in order to decide when to split a node in the Area Node Tree. This is a reactive heuristic, because we need to first collect conflict counts and then adapt based on their values after conflicts have already occurred. However, this version of the heuristic can be used transparently, without any specialization to or modification of the application code. In the second version of node splitting, we use application knowledge in order to decide when nodes need to be split. This is a proactive heuristic based on player density and location on the game map. We present an overview of node splitting and its versions based on HyTM statistics, and application knowledge, respectively, in the next three sections.

6.4.1 Selective Node Splitting Overview

Our node-splitting heuristic is designed to alleviate contention on both area node hash tables and area node tags in contended areas of the game. It achieves this by splitting area nodes that generate or are predicted to generate many conflicts, recursively into halves. This operation can alleviate conflicts due to false sharing in either data or meta-data access. In terms of alleviating conflicts due to meta-data access, this operation doubles the number of associated hash tables for storing players at each splitting step. In terms of alleviating data false conflicts, the same operation can refine player grouping to smaller regions of the game map.

When an area node is split, we create its children, and move the players from the original node into the appropriate children, depending on the players’ position on the game map.

Conversely, when we detect that there is no contention in a certain region of the map, corresponding to a specific part of the area node tree, we could, in principle, coalesce the nodes in that subtree to fewer nodes, and move the players up in the nodes that have now become leaves.

In our current prototype implementation we do not perform incremental coalescing, however. This is due to an implementation artifact in SynQuake which treats leave and non-leave nodes differently making it easier to create new leaves than retract existing ones.

With selective splitting, we use a baseline tree level of 4, and we either i) maintain leaves at this level if no contention is detected or predicted for them, or ii) we split level 4 nodes recursively until the desired level where contention is alleviated.
6.4.2 Selective Node Splitting Based on HyTM-provided Statistics

The version numbers, conflict counters and other statistics provided by the HyTM library can help detect contention in a generic, application-independent way. The application can trigger the node splitting heuristic based on information provided by the TM, such as, per-object conflict counters, and abort rates.

As we previously mentioned, the conflict counters and version tag numbers maintained by HyTM within the tag objects can be read directly by the application.

These statistics indicate contention only from the point of view of Layer 2 of our HyTM system. A conflict counter is incremented when read validation of an application object fails; the failure of read validation implies a real conflict, which in turn means that there is actual contention on that transactional object. Therefore HyTM maintained conflict counters cannot offer insights into whether or not contention occurs in hardware and on what objects, although sometimes the contention on the same objects occurs in both hardware and software. This is particularly the case if true sharing occurs on any part of the respective objects. Similarly, read validation leading to conflict counter incrementation occurs only in our hybrid transactions executing in layer 2 in our current prototype.

This is because, for the pure hardware layer (layer 1), read validation is automatically performed by the hardware at the granularity of the cache line; for the pure software layer (layer 4), we use a global lock, hence cannot experience contention. Therefore, read validation occurs only for hybrid transactions (layers 2 and 3), and the conflict counters and our current implementation bypasses layer 3 entirely.

The version tag number is incremented in either of the following two scenarios:

- For a pure hardware transaction (layer 1), or a pure software transaction (layer 4), we simply increment the version tag number.

- For a hybrid transaction (layers 2 and 3), the version tag number is incremented only at commit time, when the private write buffers are flushed after read validation has succeeded.

As before, there are two categories of data structures on which contention may occur: the application objects (player tags), and metadata objects (area nodes and hashtable tags). As we explained in Section 6.1.1 players can use their own individual tag objects, or they can use the tag object of a specific area node (e.g., the area node that they belong to).

We maintain object tags, hence conflict counters for all these types of objects (players, area nodes and area node hashtables). The information provided by the conflict counters can be used as a trigger for heuristics that perform node-splitting or node coalescing.

Our current implementation only uses the information provided by the conflict counters of hashtables, but this can easily be extended by using the conflict counters of the other types of objects in addition to, or instead of the conflict counters of hashtables. We chose to use the conflict counters for hashtables as they experience some amount of contention both for entity and for set granularities.

Then, for each leaf node, we evaluate if the respective conflict counter for that area node is above some threshold and also more than a percentage of the total conflicts generated by all area nodes. We use an absolute value for the threshold count in addition to a relative value compared to the overall count in order to avoid splitting in situations where conflict counts are low overall. In our experiments, the absolute threshold count was chosen to be 50, although further investigations yielded similar results with values of 40 or 60. In our current implementation, a node splitting operation continues to split the
newly generated nodes recursively using a pre-computed estimate of the tree depth we wish to achieve. The pre-computed estimate uses the simplifying assumption that each time we split we are halving the number of conflicts, and thus determines the number of times we need to split in order to fall under the conflict counter thresholds.

6.4.3 Performance of Node Splitting based on HyTM-provided Statistics

In this Section, we show the performance of our heuristic that performs node splitting triggered by statistics provided by the conflict counters. The conflict counters provide a numerical value of the contention experienced by a transactional object.

Figures 6.8 through 6.10 show the execution time of our conflict counter-based heuristic against the execution times of various fixed tree depths, for all three quest scenarios. Our heuristic outperforms several of the fixed tree depth approaches in all three quest scenarios, and is close to the best-performing fixed tree depths in the 1qc scenario, that experiences the highest contention.

Figures 6.11 through 6.13 show the percentage of transactions commits by layer. Again, our heuristic manages to allow more transactions to commit in the pure hardware layer and the hybrid layer than all but the best-performing fixed tree depths.

6.4.4 Selective Node Splitting Using Application Knowledge

This version of the selective node splitting heuristic is based on a combination of several application-level criteria. We decide that a node should be split when both of the following conditions are met:

- If the node has a sufficient number of players as an absolute value (this is to avoid splitting nodes with too few players).
- If a sufficient percentage of the players in the node are ”close enough” to a region owned by a different thread.

The two thresholds for the two conditions are relatively flexible, i.e., a relatively wide range of such thresholds can be used, and they produce similar results. For our experiments the two thresholds have been respectively set to 50 players, and 20% of the size of the region covered by an area node. The percentage of players that must be close to the border is fixed to 15%; our heuristic starts from an initial tree depth of 4.

6.4.5 Performance Comparison of Both Node Splitting Heuristics

Figures 6.21, 6.22 and 6.23 show the execution times for the three quest scenarios, comparing the performance of our heuristic with a series of fixed tree depths, from 4 to 16.

Figures 6.24, 6.25 and 6.26 show the transaction commit percentage per layer for the same quest scenarios.

The first three figures show that our heuristic outperforms several of the fixed tree depth scenarios, although it does not always reach the same performance as the best fixed tree depth. For the 1qc and 4qq scenarios, the heuristic outperforms tree depths of 4, and above 14. For the noq scenario, its performance is comparable to the best performing fixed tree depths.
The second set of three figures confirms the findings of the previous three graphs: our heuristic commits more transactions in the hardware layers than all others except the best performing fixed tree depths for the 1qc and 4qq scenarios, and commits a number of transactions in the hardware layers comparable to the one achieved by the best fixed tree depths (8 and 10) for the noq scenario.

Figures 6.14 through 6.19 show the execution times, and the transaction commit percentage by layer, for the same data points as above, but with the further addition of the node splitting heuristic as described in Section 6.4.4. "heur_cc" is the heuristic based on conflict counters, and "heur_ns" is the heuristic based on application-specific knowledge as described in Section 6.4.4.

The figures show that using only information provided by the TM platform, and without specific information collected from the application itself, our node splitting heuristic based on HyTM statistics shows similar performance to the node splitting heuristic with application knowledge, for the 4qq and noq scenarios, and even outperforms the heuristic using application-specific knowledge for the scenario with the highest degree of contention, 1qc.
Figure 6.8: Execution times for fixed tree depths versus the heuristic based on conflict counters, for the 1qc scenario.

Figure 6.9: Execution times for fixed tree depths versus the heuristic based on conflict counters, for the 4qq scenario.
Chapter 6. Variable Granularity in HyTM

Figure 6.10: Execution times for fixed tree depths versus the heuristic based on conflict counters, for the noq scenario.

Figure 6.11: Transaction commit percentages per layer for layers 1, 2 and 4 (from bottom to top) for the 1qc scenario. In each bar group, the first bar is for ent granularity, while the second bar is for set granularity.
Figure 6.12: Transaction commit percentages per layer for layers 1, 2 and 4 (from bottom to top) for the 4qq scenario. In each bar group, the first bar is for ent granularity, while the second bar is for set granularity.

Figure 6.13: Transaction commit percentages per layer for layers 1, 2 and 4 (from bottom to top) for the noq scenario. In each bar group, the first bar is for ent granularity, while the second bar is for set granularity.
Figure 6.14: Execution times for fixed tree depths versus the heuristic based on conflict counters (heur_cc) and the heuristic triggered by application-specific knowledge (heur_ns), for the 1qc scenario.

Figure 6.15: Execution times for fixed tree depths versus the heuristic based on conflict counters (heur_cc) and the heuristic triggered by application-specific knowledge (heur_ns), for the 4qq scenario.
Figure 6.16: Execution times for fixed tree depths versus the heuristic based on conflict counters (heur\_cc) and the heuristic triggered by application-specific knowledge (heur\_ns), for the noq scenario.

Figure 6.17: Transaction commit percentages per layer for layers 1, 2 and 4 (from bottom to top) for the 1qc scenario. In each bar group, the first bar is for ent granularity, while the second bar is for set granularity. heur\_cc is the heuristic based on conflict counters, and heur\_ns is the heuristic triggered by application-specific knowledge.
Figure 6.18: Transaction commit percentages per layer for layers 1, 2 and 4 (from bottom to top) for the 4qq scenario. In each bar group, the first bar is for ent granularity, while the second bar is for set granularity. heur_cc is the heuristic based on conflict counters, and heur_ns is the heuristic triggered by application-specific knowledge.

Figure 6.19: Transaction commit percentages per layer for layers 1, 2 and 4 (from bottom to top) for the noq scenario. In each bar group, the first bar is for ent granularity, while the second bar is for set granularity. heur_cc is the heuristic based on conflict counters, and heur_ns is the heuristic triggered by application-specific knowledge.
In this Section, we have shown the performance of our metadata heuristics compared to a range of fixed tree depths. Figure 6.20 shows the performance of the metadata heuristics over the average of the execution times of the fixed tree depths that we investigated. We believe that the range of fixed tree depths we have used, i.e., values between 4 and 16 inclusively, covers all of the reasonable fixed tree depths that could be used for a large set of execution scenarios. Our two heuristics perform 12.5% and 16.7% faster than the average of the execution times of all fixed tree depth configurations, respectively. These numbers are obtained by averaging the execution times of all fixed tree depths across all quest scenarios, and comparing with the average execution times of our heuristics across all quest scenarios.

6.4.6 Overhead of Meta-data Heuristics

We now examine the performance of our heuristic in more depth, by comparing it against a series of tree depths (including the best performing ones), while the heuristic itself starts from the same initial tree depth as the fixed one. This is in contrast with the previous set of results we showed above, where the heuristic always started from an initial tree depth of 4.

The goal of these experiments is to show that not only does our heuristic improve performance compared to other tree depths, but also that it does not have a negative impact on the performance of the program, no matter what initial tree depth it starts from.

For brevity, we only show the data for set granularity, as the results for entity granularity are similar. The same thresholds for the heuristic (50 players and 20%) were maintained as in the set of experiments we showed previously.
As Figures 6.21 through 6.23 indicate, our heuristic always has comparable or better performance relative to an execution that uses the corresponding fixed tree depth. We improve performance when the fixed starting tree depth is below the "sweet spot", that is, when the sweet spot can be achieved by further splitting the tree. We do not improve performance when starting from a fixed tree depth that’s larger than the "sweet spot" one, because our heuristic favors performing selective node splitting, and, in the cases shown, no node coalescing occurs.

It is notable that for the 4$q$ scenario, the tree depth has much less impact compared to the other two quest scenarios, due to the lack of contention. Even in this case, our heuristic always performs comparable to the corresponding tree depth, and never degrades performance.

6.5 Combining Metadata Heuristics and Data Heuristics

Previously we have shown that our data heuristics do not help performance on their own, as opposed to our meta-data heuristics. In this Section, we combine our meta-data and data heuristics and evaluate their performance in combination.

The experiments were conducted with 8000 players, on a 512x512 game map. When enabled, the metadata heuristic is applied only once, close to the start of the benchmark (allowing for sufficient cycles to accumulate meaningful values in the conflict counters), while the data heuristic is applied every cycle. In both cases, the heuristics run during the single-threaded step of the simulation.

Figures 6.27, 6.28 and 6.29 show the execution times for the three quest scenarios respectively, while Figures 6.30, 6.31 and 6.32 show the transaction commit percentage per layer for each of the scenarios.
Chapter 6. Variable Granularity in HyTM

106

Figure 6.21: Performance evaluation of our node-splitting heuristic for the 1qc scenario for set granularity, starting from various fixed tree depths. The first bar in each group represents an execution without the heuristic, while the second bar represents an execution with the heuristic enabled.

4qq set

Figure 6.22: Performance evaluation of our node-splitting heuristic for the 4qq scenario for set granularity, starting from various fixed tree depths. The first bar in each group represents an execution without the heuristic, while the second bar represents an execution with the heuristic enabled.
Figure 6.23: Performance evaluation of our node-splitting heuristic for the noq scenario for set granularity, starting from various fixed tree depths. The first bar in each group represents an execution without the heuristic, while the second bar represents an execution with the heuristic enabled.

Figure 6.24: Transaction commit percentage per layer for layers 1, 2 and 4 for the 1qc scenario. The first bar in each group represents an execution without the heuristic, while the second bar represents an execution with the heuristic enabled. For each bar, layer 1 is the bottom-most portion, whereas layer 4 is the top one.
Figure 6.25: Transaction commit percentage per layer for layers 1, 2 and 4 for the 4qq scenario. The first bar in each group represents an execution without the heuristic, while the second bar represents an execution with the heuristic enabled. For each bar, layer 1 is the bottom-most portion, whereas layer 4 is the top one.

Figure 6.26: Transaction commit percentage per layer for layers 1, 2 and 4 for the noq scenario. The first bar in each group represents an execution without the heuristic, while the second bar represents an execution with the heuristic enabled. For each bar, layer 1 is the bottom-most portion, whereas layer 4 is the top one.
Figure 6.27: Execution times for fixed tree depths versus combining metadata heuristics (cc) and data heuristic (cp) for the 1qc scenario.

Figure 6.28: Execution times for fixed tree depths versus combining metadata heuristics (cc) and data heuristic (cp) for the 4qq scenario.
Chapter 6. Variable Granularity in HyTM

Figure 6.29: Execution times for fixed tree depths versus combining metadata heuristics (cc) and data heuristic (cp) for the noq scenario.

Figure 6.30: Transaction commit percentage per layer, for layers 1, 2 and 4 (from bottom to top) for the 1qc scenario. In each bar group, the first bar is for ent granularity, while the second bar is for set granularity. cc is the metadata heuristic, and cp is the data heuristic.
Figure 6.31: Transaction commit percentage per layer, for layers 1, 2 and 4 (from bottom to top) for the 4qq scenario. In each bar group, the first bar is for ent granularity, while the second bar is for set granularity. cc is the metadata heuristic, and cp is the data heuristic.

Figure 6.32: Transaction commit percentage per layer, for layers 1, 2 and 4 (from bottom to top) for the noq scenario. In each bar group, the first bar is for ent granularity, while the second bar is for set granularity. cc is the metadata heuristic, and cp is the data heuristic.
The first four bar groups represent runs with fixed tree depths. We are showing the fixed tree depth of 4, because that is the tree depth that our heuristics start from. We are also showing the three best performing fixed tree depths for this configuration. The next two bars show runs with our heuristics enabled. The metadata heuristic, \textit{cc}, is always enabled, while the data heuristics, \textit{cp} is first disabled, and then enabled.

As seen in previous experiments, our heuristics get close to the performance of the best performing fixed tree depths, but this is not the most interesting aspect here.

The interesting part is the difference between \textit{ent} and \textit{set} for the baseline runs, as well as for the cases when the heuristics are combined.

An important thing to understand is that when the data heuristic is enabled, \textit{ent} and \textit{set} only represent the granularity that the players start from. Once the heuristics start making changes to the granularities of the players, it no longer matters what the initial granularity was. Eventually, things will converge to the same set of granularities for players, both in the case where we start from \textit{ent} granularity, and in the case where we start from \textit{set} granularity. Therefore, when the data heuristic is enabled, we should expect to see a much smaller difference between \textit{ent} and \textit{set}.

This is why, for example, when the \textit{cc} metadata heuristic is on, but the data heuristic \textit{cp} is off, there is a difference between \textit{ent} and \textit{set}, because the metadata heuristic primarily affects metadata, and it does not change the granularity of players between \textit{ent} and \textit{set}. However, as soon as we turn on \textit{cp}, the difference between \textit{ent} and \textit{set} is significantly smaller. Again, this is expected, because the data heuristic will affect players identically in both cases, regardless of what granularity the players started with.

Another interesting observation is that the difference between \textit{ent} and \textit{set} is significant for the fixed depth tree of 4, but much less for, say, the fixed tree depth of 10 or 12. To clarify, we are referring to the difference in execution time between the \textit{ent} and \textit{set} bars within the same bar group (e.g., for a given tree depth), and not between the difference in execution times between \textit{ent} and \textit{set} across different bar groups (or across different tree depths).

We believe that the reason for this is that the difference in the number of metadata tags and data tags tracked in the two situations is much smaller for a tree depth of 12. The number of tags tracked for \textit{ent} is the same in both cases. However, for a tree depth of 12, the number of metadata tags is much higher than it was for a tree depth of 4, and that’s why \textit{set} is no longer considerably worse than \textit{ent} for the case with 12 tree depth. So, the difference between \textit{ent} and \textit{set} within the same group is less, because in the case of tree depth of 4, we were tracking many data tags for \textit{ent}, but much fewer (metadata) tags for \textit{set}. With a tree depth of 12, we are still tracking the same number of tags for \textit{ent}, but we are now tracking more metadata tags for \textit{set}.
Chapter 7

Varying Granularity in Conjunction with other Platforms and Applications

Our main focus in this dissertation has been primarily on a study of variable granularity on a platform with hardware support for TM and for a game application, SynQuake.

In this Chapter, we investigate the applicability and effect of varying granularity for other applications, and in other contexts (such as a hardware TLS framework).

In Section 7.1 we investigate the impact of one of our variable granularity heuristics for other applications. Specifically, we show results for our user-guided conflict prediction heuristic which provides hints to the TM system regarding the likelihood of conflicts for specific parts of data structures or memory regions for a simple kernel application with regular and predictable memory access patterns, called Successive Over-Relaxation (SOR).

In Section 7.2 we show an investigation into the impact of variable granularity access tracking for a set of SPECint benchmarks, in the context of an emulated hardware TLS framework. Our study presents a case for fully supporting variable granularity in hardware for a wide range of granularities, such as, word, various cache line sizes, and page level.

7.1 Investigating Varying the Granularity for Other Applications

We believe that an interesting research topic would be the use of variable granularity in the context of parallel applications running under a TM, coupled with feedback or guidance from the programmer. The programmer would augment the automated monitoring and adaptive abilities of the TM with hints, or guidelines, that would further help the TM system refine or focus its optimizations in a given direction. Considering the topic of our thesis, our interests naturally lie in the direction of variable granularity, but other types of guidelines could be supported as well.

In this section, we present an example of the use of programmer guidance we discuss possible extensions to our previous techniques which enhance the interface between the application and the TM in two
ways: i) through user directives that provide guidelines to the TM for varying its tracking granularity and ii) through exposing lightweight TM statistics in an STM for selective TM monitoring and adaptation, a generic application kernel, Successive-Over-Relaxation (SOR), and we provide results for SOR that support our claims.

A natural follow-up would be an investigation of implementing support for a communication infrastructure between an application and the underlying TM library for a hardware or hybrid TM.

### 7.1.1 User-Guided Adaptations for Applications with Regular, Predictable Access Patterns

Consider an application with static, regular patterns, such as the Successive Over Relaxation (SOR) code presented in Figure 7.1. With this simple code, the programmer knows that the objects accessed are array elements in the grid array, the access pattern is regular and consists of iteratively recomputing each element as the average over its four neighbors in the grid array.

Task assignment is static, done once, at initialization. Task assignment consists in partitioning the array in blocks of array rows; the only sharing between threads is on the array elements on the rows adjacent to per-thread partition boundaries.

```c
for number_of_timesteps {
    for (i=1; i<n; i++)
        for (j=1; j<n; j++)
            temp[i][j] = 0.25 * (grid[i-1][j]+grid[i+1][j] + grid[i][j-1] + grid[i][j+1])
    for (i=1; i<n; i++)
        for (j=1; j<n; j++)
            grid[i][j] = temp[i][j];
}
```

Figure 7.1: Successive Over Relaxation (SOR) code

Due to the use of a private, per-thread temp array with copies of temp into grid on each iteration, as in the classic SOR code shown, inter-thread conflicts can be completely avoided for SOR. However, in this paper, we use this simple code as representative of a larger class of array-based, regular, iterative computation, with sharing between threads at partition boundaries. For these patterns, the programmer can provide accurate guidance to the STM about the sharing patterns for regular application patterns, by asserting that certain ranges of the array are most likely conflict-free, or conversely, asserting the range around the boundary between thread allocations within which conflicts (or even sharing) is still likely.

Because the sharing pattern is very predictable in this case, the STM can use the most aggressive granularity optimization for data that is indicated as likely not shared, i.e., tracking using a single, global, per-thread granularity for the array elements in the non-shared range. The STM uses a per-element granularity to track the shared array elements at the boundary between partitions.

### 7.1.2 Experimental Findings

For our experiments we use four 2-way Hyperthreaded CPUs at 2.8GHz with 16GB of RAM, running Linux 2.6 and gcc 3.3.6. Threads are bound to processors such that each thread executes alone on its
(Hyperthreaded) CPU. Each result represents an average over three runs.

Figure 7.2: Execution time for SOR when using fine granularity only, versus using fine granularity for the array elements on the border rows of each thread’s allocated chunk, and per-thread global locks for the non-border elements. We show data for the Read-Optimistic (RO) and Fully Optimistic (FO) conflict detection policies, with Wait-for-Readers as the conflict resolution policy.

We instrumented a version of SOR (successive over-relaxation) with libTM; the pseudocode for SOR is shown in Figure 7.1. The code does not require synchronization in terms of locking, but it does require barriers. The code partitions the matrix into equal chunks and assigns each chunk to a thread.

We ran two versions of the code: in one, each matrix element has its own lock (fine granularity); in the other, elements that are on the row adjacent to another thread’s chunk still have fine granularity, but the other elements within the same thread’s workload all share a common thread-level lock. While the code does not require border elements to actually be locked, we are showing this example to gauge the potential performance impact in an application with similar access patterns where locking would be necessary. As Figure 7.2 shows, the latter code version consistently outperforms the fine-granularity one across all conflict detection policies, by as much as 79%.

The explanation for this can be found in Figure 7.3, which shows a comparison of the cache misses registered for the two versions. As can be seen, the ratio of cache misses between the two versions follows a trend similar to that observed for the execution times.

7.1.3 Summary

In summary, SOR is a simple benchmark representative for applications with static, regular access patterns involving arrays.

We showed that a technique similar to our conflict prediction method combined with guided conflict tracking based on given predictions brings performance benefits of up to 79% for SOR, when compared to a static fine-grained tracking method. We applied more aggressive reductions of TM tracking here.
Figure 7.3: Cache misses for SOR when using fine granularity only, versus using fine granularity for the array elements on the border rows of each thread’s allocated chunk, and per-thread global locks for the non-border elements. We show data for the Read-Optimistic (RO) and Fully Optimistic (FO) conflict detection policies, with Wait-for-Readers as the conflict resolution policy.

due to the highly predictable nature of the access pattern. We also showed that the improvements are mostly due to improved cache behavior in the underlying architecture for the optimized code.

7.2 Potential of Hardware Support for Variable Granularity

In this Section we present results demonstrating the need and potential for variable-granularity access tracking in an optimistic parallelization system—specifically for a hardware-based TLS system running speculatively-parallelized SPECint applications. We assume that the hardware-based TLS system can track access at word-size, cache-line size and page size granularities, and that mechanisms by which the tracking granularity can be varied are already in place.

We show that the largest access tracking granularity that does not incur false violations varies significantly across applications, within applications, and across ranges of memory—from word-size to page size. We also show that if variable-granularity access tracking is supported, the number of memory ranges that must be tracked and compared to detect conflicts can be reduced by an order of magnitude compared to word-level tracking, without increasing false violations. Our work that investigates the potential for variable granularity has been covered in detail [8].

7.2.1 Experimental Framework

For this study we use the benchmark and simulation infrastructure developed by Steffan et al. [72] for TLS. Several SPECint benchmarks are profiled for execution time and dependences, then selected
loops are transformed for speculative execution. The compiler outputs C source code which is then compiled with gcc v2.95.2 using the “-O3” flag to produce optimized, fully-functional MIPS binaries. We use a simple in-order single-CPI multiprocessor simulator to analyze the memory access patterns of the benchmark applications. To maintain reasonable simulation time, we truncate the execution of all appropriate benchmarks by fast-forwarding the initialization portion of execution and simulating up to the first billion instructions.

It is important to understand that we are measuring regions of code that were selected for speculative execution based on profile information that assumed a 32-byte granularity for tracking reads and word-level granularity (4-bytes) for tracking writes. Throughout the rest of this section, we will refer to these regions simply as "code regions" or "speculative code regions". Also, we limit this study to only measuring “true” (i.e., Read-After-Write) dependences between speculative threads.

### 7.2.2 The Problem with a Uniform Coarse Grain Approach

![Figure 7.4: Increase in false conflicts detected relative to word-level tracking granularity (4-bytes) when tracking at typical cache-line granularities of 32, 64, and 128 bytes.](image)

To illustrate the problems of tracking accesses at a fixed, uniform, coarse granularity, Figure 7.4 shows the increase in false conflicts detected relative to a word-level tracking granularity (4-bytes) when tracking at typical cache-line granularities of 32, 64, and 128 bytes. For all applications except for JPEG the amount of false conflicts increases significantly with grain-size, with PARSER suffering the most. Furthermore, recall that the regions of code that were selected for speculative parallelization were done.
so with a tracking granularity of 32-byte cache line size in mind—hence these results are conservative since these levels of false conflicts could be tolerated while still improving performance. These results indicate that cache-line granularity access tracking is non-ideal for most applications, and that word-level tracking is desired in at least part of nearly every application.

### 7.2.3 Variance in Ideal Granularity

![Figure 7.5: Ideal granularity for all speculatively-parallelized code regions—each bar represents a code region, and the numbers in parentheses count the code regions for which the coarsest granularity ($2^{12}$ bytes, page-level) is ideal.](image)

To measure the potential for varying access tracking granularity, we compute the *ideal granularity*—meaning the coarsest granularity which does not incur any false violations. We begin by measuring the ideal granularity across all speculatively-parallelized code regions in our benchmarks, as shown in Figure 7.5. Access tracking granularities vary from word-level (4 bytes, or 2 on the y-axis) to page level (4k bytes, or 12 on the y-axis). Each bar represents a code region; to save space, the code regions for which the coarsest granularity ($2^{12}$ bytes, page-level) is ideal are not shown but are instead counted by the numbers in parentheses beside each benchmark. To clarify, a bar with height 4 represents a speculative code region for which the ideal granularity is $2^4$ bytes, meaning that there is at least one range in memory for that region for which a tracking granularity of $2^5$ bytes would result in a false conflict. Hence for this experiment we are picking only one common ideal granularity per code region.

It is apparent from the figure that no application prefers a single granularity. Most applications have
Chapter 7. Varying Granularity in Conjunction with other Platforms and Applications

Figure 7.6: Distribution of ideal granularities across all memory ranges across all speculative code regions for each benchmark. “grain X” means an access-tracking granularity of $2^X$ bytes.

at least one code region that requires the finest granularity access tracking ($2^2$, 4 bytes) to avoid false conflicts; in particular, go has a large number of code regions requiring the finest granularity. Tracking accesses at typical cache-line sizes ($2^5$, $2^6$, $2^7$ bytes, i.e., 32, 64, or 128 bytes) is sufficient for some code regions, but many require finer-granularity. There are a significant number of code regions that require only page-granularity access tracking, i.e., the region counts shown in parentheses; in particular, parser and twolf each have a large number of such code regions. This observed high variance in ideal granularity across applications and speculatively parallelized code regions clearly motivates an approach that at least allows the access tracking granularity to vary per code region.

Given that ideal granularity varies widely per code region, we are motivated to further investigate how ideal granularity varies across accessed ranges of memory. Figure 7.6 shows the distribution of ideal granularities across accessed ranges of memory, across all code regions for each benchmark—hence each segment labeled “grain X” represents an access-tracking granularity of $2^X$ bytes. To clarify, we have determined the coarsest access tracking granularity for all accessed ranges of memory such that (i) no false conflicts are incurred and (ii) no two tracking elements overlap.

Again we find that the ideal granularity varies widely. The fraction of memory ranges that require finest grain access tracking (4 bytes) is surprisingly small: less than 10% for all benchmarks except for crafty which is 23%. The fraction of memory ranges that can tolerate the coarsest grain access tracking ($2^{12}$ bytes, page-level) significant: more than 70% of tracking elements for crafty and jpeg, and more than 20% for mcf, perlbmk, and twolf. Typical cache-line size granularities ($2^5$, $2^6$, $2^7$ bytes, i.e., 32, 64, or 128 bytes) are the ideal granularity for significant fractions of tracking elements for some applications such as li and mcf, but are an insignificant fraction of others such as crafty and jpeg. The distribution of ideal granularities varies widely across benchmarks (and also across code regions according to data we do not show here), motivating an approach that can vary the granularity
Chapter 7. Varying Granularity in Conjunction with other Platforms and Applications

7.2.4 The Potential for Reducing Overhead

Access tracking at a variable granularity for speculative optimization systems can potentially reduce overheads without increasing false conflicts. In particular, using a variable-granularity approach can reduce the number of tracking elements used, which in turn can reduce tracking storage, and the traffic and latency between speculative threads or transactions resulting from conflict detection.

To provide a measure of this potential benefit, in Figure 7.7 we provide an estimate of how a variable-granularity access tracking approach can dramatically reduce the number of tracking elements required. In particular we plot the reduction in number of tracking elements when tracking granularity can vary across accessed ranges of memory (at the ideal granularity), relative to the number of tracking elements for uniform, fixed granularities of 4, 32, 64, and 128 bytes (the finest granularity as well as typical cache-line sizes).

![Figure 7.7: Reduction in number of tracking elements when tracking granularity can vary across accessed ranges of memory (at the ideal granularity), relative to the number of tracking elements for uniform, fixed granularities of 4, 32, 64, and 128 bytes.](image)

Relative to the finest-granularity case (4 bytes), the reduction factor is at least an order of magnitude for every benchmark; For JPEG, MCF, and PERLBMK the reduction factor is even more dramatic at 458, 51, and 51 times fewer tracking elements respectively (these bars are cropped in the figure). Compared to a 32-byte tracking granularity the reduction factor is still more than 3x for every benchmark. For coarser granularities the reduction factor is still significant, but tracking at these granularities will result
in undesirable increases in false conflicts as shown previously in Figure 7.4. These results are very encouraging, and motivate us to further research implementations of variable-granularity access tracking for speculative optimization systems.

### 7.2.5 Implementation Issues

The results in the previous section motivate a variable granularity approach to dependence tracking for speculative parallelism systems. Such an approach could apply to TM or TLS systems, and either hardware or software implementations.

For a hardware-based TLS system, a variable-granularity approach promises to reduce conflict-detection traffic, the latency of conflict detection (a key serialization in TLS), and possibly power consumption as well. For a lock-based STM system, a variable-granularity approach promises to greatly reduce the number of locks required saving both space and lock lookup time, while at the same time reducing unnecessary lock-contention for locks that are overly coarse-grain. In both cases, the variable-granularity approach allows the speculative parallelization system to match the access patterns of the application.

In this section we demonstrated that optimistic parallelization systems that support only a coarse and fixed granularity can suffer a significant number of false conflicts. In contrast, we also showed that systems that support a more variable access tracking granularity can potentially reduce or eliminate false conflicts while at the same time reducing tracking overheads. Different optimistically-parallelized code-regions within an application are amenable to different access tracking granularities ranging from 4 bytes to page-size. The ideal granularity similarly varies widely across ranges of accessed memory. We have shown that using variable granularity for access tracking for optimistic parallelism has significant potential for reducing overhead: in particular, that for every benchmark a variable-granularity approach can reduce the number of tracking elements by an order of magnitude over tracking at a fixed granularity. We believe that these results strongly motivate variable-granularity approaches to access tracking for optimistic parallelization systems.
Chapter 8

Related Work

In this Chapter, we discuss studies related to our topic. We start by introducing work related to our own in the more general sense (hybrid TM, multiplayer games in the context of STM), and continue with more specific instances of projects that are closely related to our work and address variable granularity.

Hardware and Hybrid TM

Wang et al. [76] investigated IBM’s support for hardware TM; they look at how the STAMP benchmark suite [55] 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.

Yoo et al. [83] 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.

Yoo et al. [83], as well as our previously published work [77], investigate the impact of retries and retry limits on overall application performance. A potential improvement to our implementation would be to adjust and tune the retry limits dynamically; Diegues et al. [21] showcased their work on self-tuning TSX with mixed results.

Our implementation of TSX and hybrid TM libraries both use 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. [9] as well as a more aggressive bloom filter based multiple fall-back lock solution [9]. 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.

Damron et al. introduced a very thorough design of a complete hybrid transactional memory [19] before HTM system implementations became commercially available. Dalessandro et al. presented a STM system called NoRec [18] that inspired many iterations of STM and hybrid TM systems design. Matveev et al. recently presented an hybrid implementation of NoRec [92] that is similar to our hybrid implementation. 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 was done on simulated results.

**STM and Multiplayer Games**

The benefits of using STM for designing algorithms with random access patterns are studied in [38]. The authors claim that STM provides scalable and easy-to-develop versions of such algorithms, and show that their approach for computing a minimum spanning forest of a sparse graph is scalable. However, the authors did not achieve significant speedups because of the overheads of their STM system.

One of the first studies regarding parallelizing sequential applications using transactional memory was [47]. The authors claimed that conflicts can be determined dynamically during runtime. The application was an algorithm for Delaunay mesh generation. However, they did not implement the STM system and their work shows no experimental results or comparison with a lock-based system.

In [6], STM is for the first time applied to parallelize a Computer Aided Design (CAD) algorithm. The authors used TinySTM to parallelize the placement step of the VPR CAD tool. Their results revealed up to 7.2x speedup compared with single-threaded STM. However, the single thread performance of STM is 8x slower than sequential VPR because of STM overheads. The authors claimed that hardware support for transactional memory would be required to overcome this cost.

Massively multiplayer games have become a popular application for multi-core processors, both in academia and industry. Abdelkhalek et al. has studied the performance of game servers and argued that it is an area that is not sufficiently researched [2]. There have been several studies on both lock-based and transactional memory-based variations of the popular open-source game Quake [1, 26, 45, 86]. Abdelkhalek et al. proposed a lock-based parallel server implementation for Quake. The results revealed that due to high contention and load imbalance resulting from fine grain interaction in the game, achieving scalable results is challenging. In [86], the authors used Intel STM [62] to parallelize an implementation of the Quake server. They claimed a large number of aborts due to limited conflict detection policies of the Intel STM. Moreover, they used a static load balancing algorithm that has a poor performance compared to dynamic algorithms. In a following paper [26], they presented a new version of STM Quake that uses coarse granularity. While the results of their newer system show reasonable scalability, overall, the performance still suffers because of the STM overhead and high abort rates.

In [45], a systematic approach was proposed to compare both the parallel STM and parallel lock-based Quake game servers. The STM approach was able to achieve better results than the lock-based one. The proposed dynamic load balancing algorithms also helped to reduce false sharing and true sharing. Our load balancing algorithms are based on previous works on distributed game servers [14, 17, 61]. All of these game-specific load balancing studies conclude that maintaining locality in the game is a necessary feature for good performance.

One limitation in all the above papers is that the conflict detection granularity in the game is determined beforehand. It may be difficult or impossible for the user to know the best granularity in advance. Moreover, for different game configurations, we might need to adjust the granularity on-the-fly to achieve optimal performance. To address these issues, in this dissertation we propose several algorithms that can dynamically change lock granularity for players during the game.

Our work uses several important mechanisms of MMOGs such as *flocking* and *interest management*, used in games typically for scalability reasons [10, 51]. While these are not specifically related to STM or TM in general, we use flocking in some of our evaluation experiments, and the interest management
is one of the elements that provides the foundation for our approach that assigns granularity to game objects based on whether they are likely to interact with other game objects or not.

Proposed hardware and software support for TLS and TM perform access tracking at a variety of different fixed granularities.

**Hardware TLS and TM** Many hardware schemes for TLS [29, 72, 75] and TM [3, 30, 56, 66] track accesses at the granularity of the cache-line size of the underlying cache hierarchy (typically 32, 64, or 128 bytes). Others demonstrate the necessity of tracking at finer granularities to reduce false violations [42, 72].

**Software TLS** The software approach to TLS proposed by Cintra et al. [15] tracks accesses at a word granularity, but they show improved performance only for applications with few or no conflicts, having eliminated false conflicts. The software approach to TLS in a Java runtime proposed by Pickett et al. [65] tracks accesses at an object granularity. While objects do vary in size, there are cases when object-granularity tracking is unnecessary, and others where tracking at a finer-grain than object-level would be beneficial. Early work by Papadimitriou and Mowry [64] explored a software TLS system that builds on a DSM-like environment, and they experimented with support for a range of access-tracking granularities. They found that the granularity for detecting dependences is a critical performance factor, and demonstrate the necessity of tracking memory at a fine granularity—however they agree that this would result in unacceptable overhead.

**Software TM** Some software approaches to TM (STMs) maintain read and write sets for each transaction at a word granularity [63, 68]. In contrast, Manassiev et al. [48] use a page-level granularity to detect conflicts in their STM system; while the actual page size can be configured for this system, it remains fixed for the duration of an application’s execution. Some implementations such as RSTM [50] and DSTM [33] use object-based conflict detection, which as discussed above is indeed a form of variable-granularity access tracking but not necessarily the granularity that matches the access patterns of the application. Other STMs are implemented by locking shared data [25, 32, 33]—in such cases the entities that can be locked can also be done so at a variety of fixed granularities. While our main focus in this paper is on variable-granularity access tracking, the general concept is also applicable to the granularity of locking in lock-based STMs.

**Hybrid TM** Some TM systems are built on a combination of hardware and software. They use a mixture of the techniques mentioned above, detecting conflicts at the cache-line level [54], word level [19], or a combination of both cache-line level (in the hardware) and object level (in the software) [43].

**Specific Related Works**

**Bulk Disambiguation** Ceze et al. [13] propose hashing the read and write addresses of speculative accesses into a representative signature. They demonstrate that a signature-based approach can reduce the traffic resulting from conflict detection, but at the cost of an increase in false conflicts due to the conservative nature of signatures.
RegionScout and RegionTracker Variable conflict granularity tracking, in itself, is not a new idea. Cantin et al. have looked at improving multiprocessor performance with coarse-grain coherence tracking [11]. Zebchuk and Moshovos have used similar approaches in Region Scout [58] and Region Tracker [85]. In their case, they improved the performance of the cache coherence protocol by grouping cache lines together, with only minimal hardware support. Another benefit of their approach was power savings due to reduced lookups in the cache. However, we believe we are among the first to perform a thorough study of the potential applications and performance of dynamic conflict granularity in the context of speculative parallelization for a realistic application.

Range Adaptive Profiling Mysore et al. proposed using a tree data structure to model "hot" and "cold" memory ranges in the context of application profiling [59]. They dynamically and adaptively adjust the depth and shape of the tree (which covers the entire memory range that is being profiled), by going to finer granularity for memory ranges that experience a lot of accesses (to get more accurate measurements) and by reducing the depth of the tree for ranges with few accesses (to reduce the overhead of tracking too many separate ranges).

In their study, Mysore et al. also provide some insights into how to avoid “ping-pong”-like cyclical shrinkings and expansions of the tree, which we have not studied in detail.

Memory Tracking Granularity Earlier work by Papadimitriou and Mowry [64] also looked at the effects of tracking granularity for memory accesses in the context of a software thread-level speculation (TLS) framework. Their conclusion was that fine granularity and high accuracy in tracking memory dependences are important factors; this study acknowledges, however, that doing this exclusively in software would likely cause unacceptable overhead. In contrast, our work aims to minimize overhead by using coarse granularity whenever possible, and using fine granularity, selectively, only when necessary.

We leverage game-state knowledge to estimate or predict potential cases where the number of conflicts incurs significant performance degradation. The algorithms we present in this paper help us decide which groups of players or areas of the game may be worth tracking at a fine granularity.

Variable Granularity Access Tracking In [49], the authors use a compiler-based approach and rely on interprocedural static program analysis to determine access granularity for different shared data structures. A subset of STAMP [55] benchmarks is used, and the authors show improvements between 1.87% and 9.15% for 4 threads, and up to 21% with 32 threads. One of the major limitations of this approach is the fact that it relies on static analysis, performed at compile time, and it cannot adapt to changes in the dataset or execution of the applications at runtime. In contrast, our scheme allows varying the granularity during runtime in an adaptive manner, based on the behaviour of the application.

Adaptive Granularity in Transactional Applications In ArTA [4], the author proposes a speculative approach to dynamically adjust the granularity of accesses to shared data structures within a transaction, based on the access pattern of the accesses from past transactions. This is done by adjusting the lock granularity for the shared data structures, and the results are shown on a subset of STAMP, using TL2 [20] as their TM library. Our approach differs in several ways: firstly, it does not necessarily rely on past behaviour to adjust granularity; secondly, we can group together any number of independent
separate data structures (i.e., players), and aggregate accesses to them regardless of their memory locations, whereas ArTA establishes granularity for groups of contiguous memory addresses. Our work also addresses variable granularity in the context of a hybrid TM that uses hardware transactional support, rather than only a software TM implementation.
Chapter 9

Conclusions and Future Work

9.1 Conclusions

This work provides an exploration into the use of variable granularity access tracking in the context of optimistic parallelization frameworks.

We explored several approaches to improve the performance of a realistic MMO-like application (SynQuake) running under a hybrid hardware and software transactional memory platform (HyTM) based on Intel’s TM hardware support, Transactional Synchronization Extensions (TSX). Our focus was to address contention and reduce overheads by varying the memory access tracking granularity.

We introduced heuristics that allow an application to group together application objects for the purposes of being tracked as a group by the TM system, and we presented mechanisms to help decide when to use these heuristics to vary the access granularities of groups of objects in the application. Some of our heuristics are based on application-specific knowledge, while others rely on using statistics that can reasonably be expected to be provided by the underlying transactional memory library. We showed that it is possible to improve both the execution performance, and the percentage of transactions that commit in hardware for our application, in scenarios with varying levels of contention.

We presented additional heuristics that rely on application-specific knowledge within the framework of a software TM system, libTM, and in this context we investigated the interaction between varying conflict tracking granularity and other TM features such as the conflict detection and conflict resolution policies.

We showed that some of our heuristics can be used for parallel applications that exhibit regular memory access patterns, if the workload is distributed among threads in contiguous and relatively coarse data partitions, with concurrent accesses by different threads only to the boundary regions of these partitions.

We performed a potential study that showed optimistic speculative parallel systems experience sub-optimal performance using only fixed coarse tracking granularity, and we showed that by using variable access tracking granularity can significantly improve performance. We also showed that ideal granularity varies widely both within and across applications.

We have also presented several ideas to further improve and refine these approaches, and we expect that implementing these would lead to additional performance benefits.

In summary, this work makes the following contributions:
• We extend an existing software-hardware hybrid TM system developed in our group (HyTM) with support for varying the tracking granularity, and heuristics to dynamically vary the access tracking granularity at run-time.

• We characterize the sources of contention in a realistic benchmark SynQuake using HyTM.

• We conduct an evaluation which shows that our enhanced HyTM system with support for variable access tracking can reduce TM overhead for SynQuake by minimizing false conflicts in areas of high contention, while reducing tracking overheads and transaction sizes.

• We study the potential of porting our heuristics for variable granularity to other speculative parallelization frameworks and to other applications. We also perform a potential study for providing full hardware support for varying the conflict tracking granularity for scientific applications in speculative parallelization frameworks.

Continued investigation on this topic would need to answer at least a few of the following research questions. Should granularity be determined by memory location or by the code location of the access (i.e., load or store PC)? Can granularity be decided on-the-fly through a dynamic, adaptive system based on iterative sampling? Can a compiler help by identifying fine or coarse grain accesses e.g., through regular section analysis where feasible? What is the best way to incorporate profile information while limiting its overheads? We hope to continue to address these questions and work on other extensions to current work in our future studies of variable-granularity TLS and TM systems.

9.2 Future Work

We believe that the techniques we used on the area node tree in SynQuake can be applied in a similar manner to other applications that use spatial search data structures (such as trees) to map various regions of space and search objects located therein. We expect that varying the granularity at which these regions are tracked through the techniques we have shown can potentially improve the usage of hardware speculative resources, and provide better overall application performance.

To this end, we are investigating two other applications: moldyn (a molecular dynamics simulation) and an N-body simulation. Spatial search data structures such as trees are a natural fit for both these types of applications, and we are looking into a way to apply some of our optimizations to them, as a proof of the general applicability of our techniques.

Finally, we are continuing our investigation into orthogonal factors which can also improve conflict rates or conflict durations besides variable granularity conflict tracking. The load balancing algorithm, and the conflict detection algorithm in the TM are complementary to most of our conflict granularity adaptations, hence can be used in conjunction with them for achieving additional performance improvements. In this work we used only static load balancing algorithms because i) it was the ideal load balancing to use for the quest scenarios investigated in the game and ii) in order to simplify our analysis and characterization of contention factors that impact the performance of our game application.

However, a dynamic load balancing algorithm that is aware of potential conflicts in the application can distribute tasks to threads in such a manner as to minimize the number of conflicts on objects between threads e.g., by minimizing the number of boundaries between data partitions or object allocations to threads. Likewise, a TM protocol can optimize the conflict duration, e.g., by detecting conflicts early
in the transaction, or by making copies of contended objects thus allowing concurrent writers, or by
holding any underlying locks, as little as possible. Whether or not our granularity tracking heuristics
can be integrated more closely with the load balancing algorithm of the application and with the conflict
detection used by the TM is the object of our continued investigation.

We are also investigating other user-driven speculative optimizations, where the programmer can
instruct the TM to use increasingly aggressive optimizations in areas where few or no conflicts or data
sharing are known to occur, and/or can directly provide application-specific dependency checks to be
validated at commit.
Bibliography


[57] Katherine L. Morse et al. Interest Management in Large-scale Distributed Simulations.


