Programming, Debugging, Profiling and Optimizing Transactional Memory Programs

Ferad Zyulkyarov

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya - BarcelonaTech

A thesis submitted for the degree of

Doctor of Philosophy

June 1, 2011

I would like to dedicate this thesis to my mother who always supported me in my studies and taught me to love the science.

Acknowledgements

I would like to thank my advisors Mateo Valero, Osman Unsal and Adrian Cristal for their guidance and support during my studies. Without their help I would not be able to make these research contributions and at all understand what indeed research is. In equal way, I would like to acknowledge and thank Tim Harris who in fact has been like my external advisor. During my PhD Tim was providing me with invaluable support on Bartok, helping me in writing papers, and above all he was a constant source of inspiration.

I would like to thank Eduard Ayguadé for his advice on transactifying the Quake game. Transactification of the Quake game server constitutes large part of my thesis and also it has motivated the other ideas in my PhD work.

I would like to specially thank Mateo Valero for his involvement in the procedures that have allowed this thesis become a reality. Also, I would like to acknowledge his dedication and continuous effort for making BSC a platform of research excellence from which I was benefiting in various ways.

Of course, many thanks go to my friends Srdjan Stipic, Sasa Tomic, Valdimir Gajinov, Vesna Smiljkovic, Javier Arias, Nehir Sonmez, Otto Pflucker, Adria Armejach, Oriol Prat, Oriol Arcas, Neboyjsa Miletic, Gokcen Kestor, Gulay Yalcin, Vasilis Karakostas, Milovan Djuric, Milan Stanic, Tim Hayes, Ivan Radkovic, Nikola Bezanic, Vladimir Marjanovic, Nikola Markovic. Surely without them this dissertation would not be complete and the time during my PhD would not be that great fun.

Last but not least I would like to acknowledge the institutions which has been directly or indirectly supporting my PhD work. For the completion of this work I have used resources provided to me by my employer, the Barcelona Supercomputing Center (BSC-CNS), and by the Department of Computer Architecture at the Universitat Politécnica de Catalunya-BarcelonaTech, where I have pursued my PhD.

Several institutions have provided additional funding for this project. I was supported by the FI scholarship from the Catalan Government. This work was supported by the cooperation agreement between the Barcelona Supercomputing Center – National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and by the European Commission FP7 project VELOX (216852).

Abstract

The shift from developing powerful monolithic CPUs to a less powerful but multi-core CPUs made developers to rethink their approach of writing programs. Programmers cannot anymore expect that their programs will execute faster on the next generation CPUs unless their programs are parallel. For many years, researchers have been seeking for various solutions to make parallel programming for shared memory architectures easier and also efficient. Transactional Memory (TM) is one such potential solution. In TM synchronizing access to shared data is simpler than locks. The programmer defines the critical sections using `atomic` blocks and the underlying TM implementation automatically executes the enclosed instructions atomically and in isolation. In contrast, when using locks, the programmer manually implements the atomicity and isolation for the shared data. In addition, when conflicts are rare, the speculative execution of `atomic` blocks promises to deliver performance which is comparable to efficient lock-based implementations. To answer the questions "Is programming applications using `atomic` blocks easier than locks?" and "Is the performance of TM competitive with locks?" we have developed a real TM application - AtomicQuake. To implement AtomicQuake, as a base we used a parallel lock-based Quake game server and replaced all lock-based critical sections with `atomic` blocks. We have found out that developing applications with `atomic` blocks would be easier than locks but the performance of STMs should be improved. In addition, the experience of developing AtomicQuake revealed unsought problems which showed that TM is not yet ready for use in production quality software. Some of these problems were related to

the language level integration of TM and other problems related to the lack of TM support in the software development tools such as debuggers and profilers. While developing AtomicQuake it was extremely difficult to debug errors and almost impossible to profile the TM relevant bottlenecks. This last problem motivated us to investigate how to extend current debuggers to debug TM applications and appropriate profiling techniques that would reveal the bottlenecks in the TM applications. We have introduced three new approaches to debug TM applications. First, the user can debug at the level of `atomic` blocks. In this approach, an `atomic` block is treated as a single instruction and the implementation details of the `atomic` blocks, weather TM or lock inference, are hidden to the user. Second, the user can debug at the level of transactions. In this approach, the implementation of `atomic` blocks is assumed to be TM and the user can step inside `atomic` blocks and examine the TM state. Third, the user can manage the TM state at debug time which is analogues to the mechanisms how one can change the CPU state. Also, we have introduced new abstractions such as debug time `atomic` blocks and TM watch points. Debug time `atomic` blocks let the user create and remove `atomic` blocks at debug time. We have implemented our ideas in an extension for WinDbg debugger and the ahead-of-time C# to x86 Bartok compiler.

To profile TM applications we have introduced new techniques that provide in-depth and comprehensive information about the wasted work caused by aborting transactions. We have explored three directions: (i) techniques to identify multiple conflicts from a single program run, (ii) techniques to describe the data structures involved in conflicts by using a symbolic path through the heap, rather than a machine address, and (iii) visualization techniques to summarize which transactions conflict most. To demonstrate the effectiveness of these techniques we have built a standalone profiling tool and a lightweight profiling framework for the Bartok compiler. The profiling framework processes the data offline or during garbage collection

thus having minimal probe effect (less than 1%) and overhead (less than 14%).

Using the profiling tool we have analyzed and optimized several applications from the STAMP benchmark suite. The profiling techniques effectively revealed TM-specific bottlenecks such as false conflicts and contentions accesses to data structures. The discovered bottlenecks were subsequently eliminated with TM-specific optimizations which target is to reduce the number of aborts and wasted work incurred by these aborts. Among the optimization highlights are the transaction checkpoints which reduced the wasted work in Intruder with 40%, decomposing objects to eliminate false conflicts in Bayes, early release in Labyrinth which decreased wasted work from 98% to 1%, using less contentions data structures such as chained hashtable in Intruder and Genome which have higher degree of parallelism.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past most of the CPUs had only one core and most of the programs developed for them were sequential. The performance of these programs directly benefited from the improvements made in the new CPUs. For example, the same program executed faster on the new CPU simply because the new CPU had higher clock frequency or it had various architectural improvements. However, during the last decade manufacturing of CPUs reached an inflection point when the industry made a turn toward developing multi-core chips instead of developing more powerful single core chips [44; 90]. The main reasons behind this change stood the power wall and the level of architectural complexity which modern CPUs reached. It was not anymore feasible to operate at higher clock frequency and the achieved performance gains in new CPU architectures were marginal compared to their complexity. After this shift program developers could not expect their programs to run faster on the next generation CPUs unless they are parallel.

One of the most popular parallel programming styles is composed of multiple streams of instructions called threads. Multi-core CPUs can execute multiple threads concurrently and potentially double the program performance by doubling the number of cores. The two most prevalent ways in which program parallelism can be expressed are *d*ata parallelism and *t*ask parallelism. In data parallelism threads perform the same set of operations on a large amount of data which is exclusively partitioned between the threads [63]. Typically such applications have no inter-thread synchronization (i.e. embarrassingly parallel) or have

very simple barrier like synchronization. Conversely, in task parallelism each thread executes different set of operations and coordinate their progress through explicit synchronization. In shared memory programming model, which is is best suited for the today's multi-core architectures, typically the synchronization in task parallel programs is implemented with mutual exclusion by using locks or semaphores. Implementing coarse grain synchronization with few global locks is easy, however such applications have poor performance and scalability. On the other side, implementation of correct and efficient fine-grain lock-based synchronization is difficult; the programmer should manually manage the locks by associating them with the shared data structures and also take special care on the order of acquiring and releasing the locks to avoid deadlock.

Transactional memory (TM) is an alternative mechanism for implementing synchronization in shared memory architectures [60]. Compared to locks, TM abstracts the complexity of implementing parallel programs. The programmer needs only to declare the atomic regions in the code and the underlying TM system transparently provides the atomicity whereas when using locks the user has to manually implement the atomicity for the operations that mutate the shared data. Typical implementations of TM execute transactions optimistically, detecting conflicts which occur between concurrent transactions, and aborting one or other of the involved transactions [56]. In applications with low contention, optimistic transactional execution delivers better scalability and performance comparable to locks. However, the ease of programming using TM is not for free as it incurs single threaded overhead and overhead on transaction aborts.

This work studies the development of software using transactional memory from programmers' point of view. Unlike most of the existing research which is focused on evaluating and improving the performance of the TM implementations this work is focused on evaluating and improving the development process of applications implemented with `atomic` blocks and TM. Its goals are to understand the effort of developing transactional programs, to report for the issues of using TM in real applications and to try to address these issues. More specifically it is motivated by providing answers to the following questions:

- Is developing parallel programs with `atomic` blocks and TM easier than locks?

- Is the performance of TM in real complex applications comparable to the performance of fine-grain lock-based implementations?
- Are existing development tools (e.g. compilers, debuggers, profilers, etc.) ready for developing transactional applications?
- How to extend existing development tools to support `atomic` blocks and TM?

To answer these questions this work starts by investigating how to use `atomic` blocks to implement the thread synchronization in a real application – AtomicQuake. Compared to the existing micro-benchmarks and small kernel applications which are developed solely for evaluating TM implementations, AtomicQuake has richer synchronization instances which exercise the corner cases for TM (e.g. error handling, I/O, failure atomicity etc.). Subsequently it continues in three main directions. First, it investigates how to extend the existing debuggers to support `atomic` blocks and TM. Second, it investigates relevant profiling techniques which provide comprehensive and in depth information about the TM specific bottlenecks in transactional applications. Third, it investigates how to optimize the transactional applications based on the obtained profiling information by leveraging the underlying TM implementation mechanisms.

1.1 Thesis Statement

In this thesis I make the following assertions regarding TM:

1. Parallel programming using `atomic` blocks is easier than fine-grain locking schemes. Programming with `atomic` blocks resembles coarse-grain locking approach. When there are many shared objects, the individual synchronization of each object or the implementation of a region based synchronization is straightforward using `atomic` block and TM. On the other side, similar fine-grain synchronization with locks require careful use of locks to avoid data races and deadlocks. Also, maintenance of code with `atomic` blocks seems to be easier than lock-based code because the concurrency is expressed through the programming language;

2. TM is not a bottleneck for the scalability of the parallel applications. However, unlike the performance results obtained with micro-benchmarks and small kernel applications, TM is not as efficient as locks in large real world applications. In a real transactional application TM has high single threaded overhead and unanticipated abort overheads at the presence of contention;

3. Current TM technology is not mature enough to be used for developing production software because of the following reasons:

   (a) Language extensions and semantics are not expressive enough to implement I/O, errors and recover from errors inside transactions. For example, locks cannot be replaced directly because their use do not match the block based structure of `atomic` blocks;

   (b) Existing application development tools such as compilers, debuggers and profilers have minimal or no support for TM. For example, debuggers are not aware of `atomic` blocks and they cannot execute `atomic` blocks atomically. Also, existing profiling tools do not provide relevant information to discover and understand the TM overheads;

4. In large parallel applications replacing the lock-based synchronization with `atomic` blocks is not straightforward. It requires careful examination of the code to understand the locking policy (i.e. which lock protects which shared data);

5. It is difficult to find synchronization errors in TM applications and debug wrong code inside `atomic` blocks because conventional debuggers are not aware of `atomic` blocks and TM. To find the synchronization errors between `atomic` blocks such as atomicity violations and asymmetric data races debuggers need to be extended with the atomicity semantics of transactions. To debug wrong code inside `atomic` blocks without observing speculative updates from other transactions, debuggers need to be extended with the isolation semantics of transactions;

6. TM applications have different types of bottlenecks which are specific to the TM programming model. These bottlenecks are caused by the aborting transactions and are difficult to anticipate and understand. To find and understand these bottlenecks properly requires new profiling techniques which report results in an from independent of the underlying TM implementation;

7. The performance of TM applications can be improved with TM-specific optimizations which leverage the specific mechanisms provided by the underlying TM implementation. For example, the same program can execute faster if the programmer uses transaction checkpoints, nested `atomic` blocks or early release.

These assertions will be demonstrated by:

1. Developing a real parallel application, called AtomicQuake, from an existing parallel lock-based version of the Quake game server by replacing all lock-based synchronization with `atomic` blocks and porting transactional applications from the STAMP TM benchmark suite from C to C#;

2. Developing a debugger extension to support debugging applications that use `atomic` blocks and TM;

3. Building a lightweight profiling framework for Bartok-STM and a profiling tool to profile TM applications.

4. Optimizing applications from the STAMP TM benchmark suite based on the obtained profiling information by leveraging the available TM-specific mechanisms.

1.2 Contributions

This work contributes to the research in TM in a number of ways:

1. Investigation of the use of `atomic` blocks and in real complex parallel program - Quake game server *(research contribution)*;

2. Demonstration that developing complex parallel programs with TM is easier than locks *(research contribution)*;

3. Demonstration that TM is not yet a mature technology for developing production software: more research is required in language integration, compiler implementation, integration into development tools such as debuggers and profilers *(research contribution)*;

4. Evaluation of a TM system and its language integration using a real application *(research contribution)*;

5. Development of A real transactional application, AtomicQuke, to drive the research in TM *(development contribution)*;

6. Design and development of a highly configurable synthetic TM workload, WormBench, used for TM stress test and modeling the transactional behavior of real applications and also rarely occurring pathological cases *(research and development contribution)*;

7. Development of C# versions of the applications from the STAMP TM benchmark suite *(development contribution)*;

8. Investigation of new debugging principles and abstractions for transactional applications*(research contribution)*;

9. Development of debugger extensions for transactional application for WinDbg and Bartok-STM *(development contribution)*;

10. Investigation of new profiling techniques for transactional applications *(research contribution)*;

11. Investigation of techniques to relate conflicting instructions to source code *(research contribution)*;

12. Investigation of techniques to related conflicting memory addresses to variable and object names from the source code *(research contribution)*;

13. Development of a stand alone profiling tool for TM applications which processes and visualizes raw profiling data *(development contribution)*;

14. Development of a lightweight profiling framework for Bartok-STM that logs runtime information about the transactions' progress and data contention *(development contribution)*;

15. Investigation of techniques for methodological optimization of transactional applications by leveraging the mechanisms available in underlying TM system *(research contribution)*;

16. Optimizing applications from the STAMP TM benchmark suite *(development contribution)*.

1.3 Publications

The work reported in this dissertation led to the following publications:

- F. Zyulkyarov, S. Stipic, T. Harris, O. Unsal, A. Cristal, I. Hur, M. Valero, *Profiling and Optimizing Transactional Memory Applications*, to appear In Proceedings of 19th International Journal of Parallel Programming (IJPP'2011) [138] (see Chapter 6.

- F. Zyulkyarov, S. Stipic, T. Harris, O. Unsal, A. Cristal, I. Hur, M. Valero, *Discovering and Understanding Performance Bottlenecks in Transactional Applications*, In Proceedings of 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'2010) [137] (see Chapter 5.

- F. Zyulkyarov, T. Harris, O. Unsal, A. Cristal, M. Valero, *Debugging Programs that use Atomic Blocks and Transactional Memory*, In Proceedings of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'2010) [136] (see Chapter 4).

- F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguade, T. Harris, M. Valero, *Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server*, In Proceedings of 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'2009) [135] (see Section 3.3).

- F. Zyulkyarov, S. Cvijic, O. Unsal, A. Cristal, E. Ayguade, T. Harris, M. Valero, *WormBench - A Configurable Workload for Evaluating Transactional Memory Systems*, Workshop on Memory Performance: Dealing with Applications, Systems and Architecture (MEDEA'2008) [134] (see Section 3.4).

- F. Zyulkyarov, O. Unsal, A. Cristal, M. Valero, *Synthetic Workloads for Transactional Memory (Poster)*, Advanced Computer Architecture and Compilation for Embedded Systems (ACASES'2007) [133].

During the work on this dissertation, the following papers were also published by the author on closely related topics:

- V. Gajinov, F. Zyulkyarov, O. Unsal, A. Cristal, E. Ayguade, T. Harris, M. Valero, *QuakeTM: Parallelizing a Complex Serial Application Using Transactional Memory* In Proceedings of 23rd International Conference on Supercomputing (ICS'2009) [42].

- F. Zyulkyarov, M. Milanovic, O. Unsal, A. Cristal, E. Ayguade, T. Harris, M. Valero, *Memory Management for Transaction Processing Core in Heterogeneous Chip-Multiprocessors*, Workshop on Operating System Support for Heterogeneous Multicore Architectures (OSHMA'2007) [132].

- M. Milovanovic, O. Unsal, A. Cristal, S. Stipic, F. Zyulkyarov, M. Valero, *Compiler Support for Using Transactional Memory in C/C++ Applications*, Workshop on Interaction between Compilers and Computer Architecture (INTERACT'2007) [85].

Except QuakeTM as presented in [42], the work reported in the remaining related papers is not directly discussed in this dissertation.

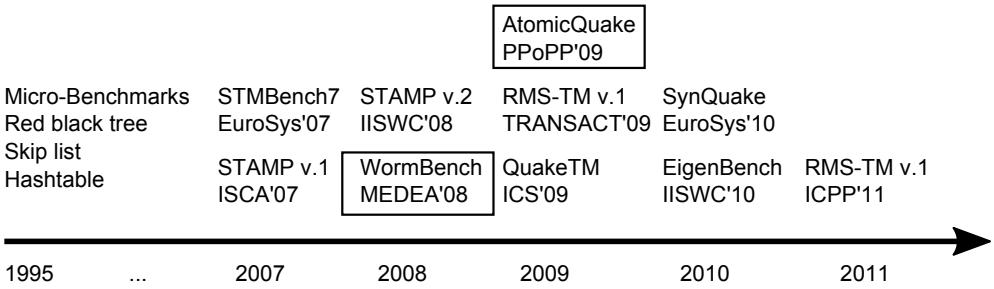


Figure 1.1: Time line of TM workloads. Only the TM workloads closest to this research are shown. My publications are surrounded with a box.

1.4 Research Context

This section makes a short overview on the closest related work to mine and sets a research context in which my research was carried. Later on, each chapter includes a more detailed related work section for its topic.

1.4.1 AtomicQuake

My research started in 2006 and spanned in a time frame of 4,5 years. During this time TM was an active research topic (and I think it is still quite active) because it was seen as a remedy for the problems around locks and promised to make synchronization implementation in parallel programming easier and yet efficient. There were proposed various TM implementations and the research was mainly focused on evaluating their performance. Until 2007 when the first version of STAMP [86] TM implementations were evaluated with micro benchmarks such as red black tree, hashtable and skip list (see Figure 1.1). Microbenchmarks are good for evaluating the TM system's implementation details such as the size of the internal data structures or cashes. However, they are not representative for evaluating the overall TM system in a setting of a real application which does real work while operating on the basic structures.

STAMP is a suite of kernel applications each implementing an algorithm with different characteristics in terms of how long they spend running inside transactions, how large those transactions are, and how likely concurrent transactions are to conflict with one another. Unlike microbenchmarks, STAMP applications

were more representative and could evaluate wider spectrum of TM implementations. STAMP applications were closer to real applications because transactions in these applications did not simply access data structures but also performed useful work. Nevertheless, the goal of STAMP was to benchmark TMs but not to evaluate how easy is to program real-world complex parallel programs using TM. Also, it was not clear how things would look like if all these algorithms implemented by individual STAMP applications are in one large application.

To study the programability aspects of TM I chose to work on the parallel version of the Quake game server [2] and this study was the first of its kind. The Quake game server was interesting for my research because it was using complicated fine grain synchronization for the irregular data structures and this synchronization resulted in 22% of the total execution when the server was fully loaded. The desired results from this work were to show that programming with TM is easier than locks and TM outperforms the lock-based implementation (i.e. to reduce the time spent for synchronization). In the published paper about AtomicQuake in 2009 [135] (see Figure 1.1), I demonstrated that programming with TM is indeed easier than locks but the performance of TM is not in par with the lock-based implementation.

While working on AtomicQuake I have encountered various problems relevant to different aspects of TM such as language primitives, semantics, I/O, library calls, error handling, development tools like compilers, debuggers and profilers. For example, I was able to compile and run AtomicQuake almost half a year after I finished it. The reason for this lag was the fact that AtomicQuake there was not a robust compiler to compile it and a TM to run it. Many times I had to re-implement basic library calls such as `sprintf`. Also, it was practically impossible to profile AtomicQuake and understand the reason for the poor performance – the TM implementation was closed source and did not provide detailed runtime information. Nevertheless, the study on AtomicQuake gave important feedback to the TM research community and opened new problems which were subsequently addressed by different researchers and also myself. This work also delivered to the research community a complex real-world TM application which could evaluate TM across the complete software stack. Even today (at the time

of writing this thesis), because of its complexity, there are few compilers and TM implementations which are robust enough to compile and run AtomicQuake.

Close works to AtomicQuake were done by Gajinov *et al.* [42] (QuakeTM) which was published the same year 2009, and Lupei *et al.* [77] (SynQuake) published 2010 (see Figure 1.1). QuakeTM was done concurrently with AtomicQuake by my colleague. The motivation of developing both AtomicQuake and QuakeTM independently was to see respectively how migrating legacy lock-based applications to TM would look like and how developing TM applications from the scratch would look like. Because QuakeTM was developed with TM in mind, there were fewer problems relevant to the TM language level primitives such as acquiring and releasing locks in non-block structured way. But still, there were common problems such as library calls, error handling and I/O. Like AtomicQuake, the study on QuakeTM confirmed that developing parallel programs with TM is easier than locks and that the performance of STM is low.

SynQuake is a stripped version of the Quake game server which includes only the main data structures and the essential features of Quake and excludes other secondary elements such as 3D space, network communication, etc. Opposite to the findings in AtomicQuake and QuakeTM (high abort rate and STM overhead) SynQuake showed performance which is competitive to a lock-based version of the same game engine. The performance improvement is achieved through an important TM and game specific optimization and also STM extensions which are tailored to the game's logic. To reduce the conflicts SynQuake implements dynamic locality-aware assignment of tasks to threads. Because of the lower conflict rate SynQuake shows better scalability and performance over AtomicQuake and QuakeTM. Also, another reason for the better performance is of SynQuake could be due to the manual instrumentation of the STM library calls in the code. AtomicQuake and QuakeTM these STM instrumentations were automatically performed by the compiler. The compiler may not be able to always determine when it is safe to skip instrumenting certain memory operations.

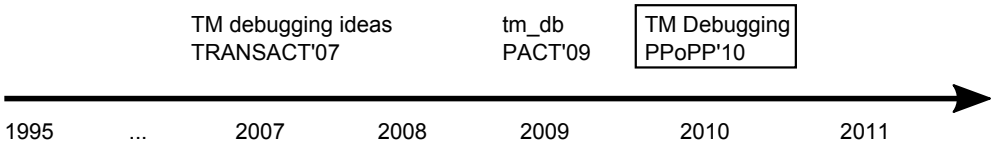


Figure 1.2: Time line of research in debugging support for TM applications. My publications are surrounded with a box.

1.4.2 WormBench

WormBench is a parameterized synthetic TM workload which I developed during the same time when I was working on AtomicQuake [134]. WormBench was designed with the goal to be easily configurable, so that it can have different TM characteristics such as abort rate, read/write set size, transaction length, etc. In this way with special configurations, WormBench can be used in stress tests such as stressing read/write set buffers or conflict detection and resolution system. Also, it is possible to prepare a configuration in which WormBench models a real TM application by having similar transactional characteristics.

Most similar TM workloads to WormBench are STMBench7 [46] published before WormBench andand EigenBench [64] published after WormBench (see Figure 1.1). Both STMBench7 and EigenBench are configurable but do not perform dummy operations inside transactions. Also, unlike WormBench, STMBench7 and EigenBench do not implement different synchronization scenarios such as producer consumer and barrier synchronization. STMBench7 has very large transactions with large read/write sets. This makes it difficult to prepare configurations which have wide range of transactional characteristics. Like WormBench, EigenBench can be configured to model a real transactional application. To some extend, configuring EigenBench is easier than WormBench because EigenBench does not perform any dummy operations but simply accesses a set of shared objects.

1.4.3 Debugging

After the work on AtomicQuake, I was highly motivated to study how to extend existing debuggers with support for debugging applications which use `atomic`

blocks and TM. While me working on AtomicQuake me and also my colleague Gejinov *et al.* working on QuakeTM had very bad experience in finding synchronization errors and debug wrong code inside atomic blocks. In this work I have introduced new debugging principles and abstractions which would help debugging transactional applications. According to the new debugging principles the user is provided with higher level and lower level view on the application. In the first approach the debugger is extended with the atomicity and the isolation properties of transactions and the underlying implementation of `atomic` blocks is abstracted – whether implemented with TM or lock-inference. In the lower level view, the underlying implementation of TM is exposed and the user has complete view on the internals. The new debug-time transactions are handy abstractions demonstrated to be useful in identifying synchronization errors. Other abstractions such as TM watch points allow the user to monitor the changes in the TM state.

In a parallel work with mine, Herlihy and Lev have developed an infrastructure for debugging transactional applications–tm_db [59] which was published just before my work(see Figure 1.2). From a user's perspective, compared to our work, when debugging a transactional application with the abstractions that Herlihy and Lev introduce, it will look like debugging at the level of transactions (discussed in Section 4.5). The primary focus of tm_db is to consistently expose the TM state through the debugger without changing the existing debugging conventions. In tm_db Herlihy and Lev introduce important concepts such as logical value, scopes, distinction between transactional reads, writes and their respective conflict coverages. These new concepts abstract the internal organization of different STM systems. Logical values are necessary for preserving the isolation property of transactions when debugging at the level of transactions. Abstracting the reads and writes with their respective coverages hides the internal mechanism to manage the read and write sets and also help in identifying false conflicts. Incorporating these new abstractions into our extension would provide users an uniform view to the TM state when debugging at the level of transactions.

In earlier work, before tm_db, Lev and Moir discussed how the debugger and the TM implantation should by integrated [73]. They surveyed features that a debugger could provide by leveraging the underlying TM system. From their work,

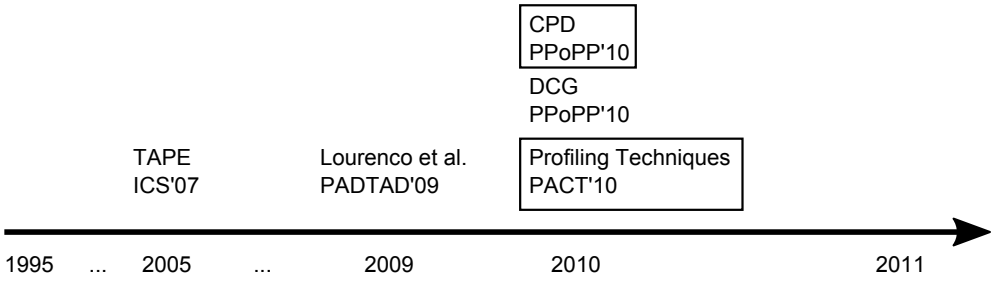


Figure 1.3: Time line of research in profiling support for TM applications. My publications are surrounded with a box.

we were inspired that seeing the read set and write set of transactions can help to understand the reason for aborts. However, a practical application of this feature showed to not be effective because conflicts happen in high amount at many different places. The lesson learnt was that we need appropriately summarized results in order to effectively reason about conflicts. After realizing this fact, I continued with the investigation of such appropriate summaries which lead to the discovery of mechanism called conflict point discovery. Conflict point discovery is summarized in the next section.

1.4.4 Profiling

AtomicQuake had poor performance but it was practically impossible to understand why. To compile and run AtomicQuake a closed source compiler and an STM implementations were used. The tool chain did not report any detailed runtime information, except the total number of aborts and read/write set size report per `atomic` block and complete program execution. It was not clear, where transactions abort, why they abort, which transactions are involved in conflicts, which shared objects are involved in conflicts, and whether conflicts are true or false. In QuakeTM, Gajinov *et al.* [42] used an adhoc approach to exploit and get very limited information about the internals of the TM. The information he could get was vague approximation about the places where transactions might conflict and respectively abort. Also, Gajinov *et al.* reported for several cases of false conflicts.

The overall experience of understanding the performance of AtomicQuake was the main motivation for investigating new profiling techniques that would provide comprehensive information about the conflicts, aborts and wasted work in a format which is independent from the underlying TM implementation. The first part of this investigation introduced basic conflict point discovery which was published at PPoPP in 2010 [136] (see Figure 1.3). Basic conflict point discovery introduced a technique to efficiently find where transactions conflict and map these places to the source code. This idea was subsequently explored much rigorously and introduced three profiling techniques in a paper which was published in 2010 [137]. The first technique identifies multiple conflicts from a single program run and associates each conflict with contextual information. The contextual information is necessary to relate the wasted work to parts of the program as well as constructing the winner and victim relationship between the transactions. The second technique identifies the data structures involved in conflicts, and it associates the contended objects with the different places where conflicting accesses occur. The third technique visualizes the progress of transactions and summarizes which transactions conflict most. This is particularly useful when first trying to understand a transactional workload and to identify the bottlenecks that are present. The discovered ideas were implemented in two program components – a profiling framework for Bartok-STM and a standalone program which process and reports the profiling data. The key achievement in the implementation is that the profiling framework has very low probe effect (i.e. does not interfere the actual execution) and marginal overhead. The low probe effect and overhead were result of successfully combining data collection with garbage collection and subsequently processing this offline.

A close work to mine was published by Chakrabarti in a poster at PPoPP in 2010 [25]. In this paper Chakrabarti introduced dynamic conflict graphs (DCG). A coarse grain DCG represents the abort relationship between the `atomic` blocks similar to aborts graph in [137]. A fine grain DCG represents the conflict relationship between the conflicting memory references. The fine grain DCG could be a nice extension to my work combined with the contextual information for the conflicts. Such information would be found useful in linking the symptoms of lost performance to the reasons at finer statement granularity.

Another similar work was done by Lourenço *et al.* published in 2009 [76]. Lourenço *et al.* have developed a tool for visualizing transactions similar to mine. They also summarize the common transactional characteristics that are reported in the existing literature such as abort rate, read and write set, etc. over the whole program execution. My work complements theirs by reporting results in source language such as variable names instead of machine addresses. Also, we provide local summary which is helpful for examining the performance of specific part of the program execution.

Neither Chakrabarti nor Lourenço *et al.* do not discuss how to identify multiple conflicts from a single profiling, how to identify conflicting objects and how to report results in source language code.

1.5 Outline

Chapter 2 provides background on transactional memory and introduces the basic concepts used throughout the text. First, it introduces the transactions as an alternative method for synchronization and explains their advantages over locks. Then it describes the typical API and the language extensions that the programmer can use to develop transactional programs. At the end follows a discussion about the design decisions involved in building a TM and various implementations in software and hardware.

Chapter 3 describes an experience of developing a real application, AtomicQuake, by replacing all lock-based synchronization in an existing parallel Quake game server with `atomic` blocks. This chapter first surveys the related work about transactional workloads and motivates the reason for using real applications for the end-to-end evaluation of TM. Then it reports for the effort of porting legacy lock-based code to TM by discussing when synchronization with TM is easier than locks and the challenges of replacing locks with `atomic` blocks. At the end follows the runtime analysis of AtomicQuake which suggest that AtomicQuake has rich transactional characteristics making it a good workload for evaluating TMs.

Chapter 4 presents new principles and approaches for debugging transactional applications. It describes how to extend the existing debuggers with the atomicity and isolation semantics of `atomic` blocks. This approach hides the underlying implementation details of the TM system. On the other side, the debugger can expose the speculative state which is maintained by the TM system. There are introduced new debugging abstractions which let the programmer change the scope of `atomic` blocks and manage the transactions at debug-time. The chapter also discusses the implementation of these features in a WinDbg debugger extension for applications compiled with Bartok compiler.

Chapter 5 presents new profiling techniques for transactional applications. Transactional applications have different types of bottlenecks which are specific to the TM programming model. The profiling techniques described in this chapter are developed to help the programmer to discover and understand these bottlenecks. These techniques can visualize the progress of the transactions and clearly show the parts of the program execution when transactions abort. Other techniques help in discovering the statements and the objects involved in conflicts. The chapter also describes the implementation of lightweight profiling framework to collect runtime data and a stand alone profiling tool which process and visualizes the runtime data.

Chapter 6 explores new techniques and approaches for optimizing transactional applications based on available profiling information. The goal of optimizing a transactional application is to reduce the contention. The contention can be reduced by decreasing the scope of `atomic` blocks, using different data structures which cause less conflicts, and reducing the shared state. Also, this chapter describes how to optimize applications by using mechanisms such as transaction checkpointing, nested transactions, early release, transaction scheduling etc. which might be available at the underlying TM system. The effectiveness of these techniques is demonstrated by first profiling and then optimizing applications from the STAMP TM benchmark suite.

Chapter 7 concludes and discusses future work that can extend the work described in this dissertation.

Chapter 2

Background - Transactional Memory

Transactional memory is an optimistic concurrency control mechanism for synchronizing access to shared data in multi-threaded programs. Analogues to the database transactions TM allows multiple threads to perform series of memory operations atomically. As an alternative to locks, TM attempts to simplify parallel programming by requiring to only identify the atomicity for the shared variables in the program. On the other side, when using locks the programmer should first identify the shared shared variables and manually implement atomicity for the operations which mutate them. Other important problems of locks are deadlocks and composability. For example, to avoid deadlocks, developers should either use special policies to acquire or release lock (i.e. two-phase locking) or detect and resolve deadlocks before each lock acquisition. Building and maintaining such lock-based programs is extremely difficult because the association between the shared variables and locks is not defined by the programming language but the programmer itself. Moreover, building complex programs from composable black-box-like components is practically impossible. To highlight the advantages of TM over locks Grossman makes an analogy between TM and garbage collection [43]. According to this analogy TM would make the implementation synchronization easier as garbage collection made memory management easier. Besides the simpler programming interface, TM can extract more parallelism by

executing transactions optimistically thus achieving better performance so long as transactions do not perform conflicting accesses.

This chapter gives background on transactional memory and introduces the basic concepts which are used later in the text. Section 2.1 presents TM from the programmers perspective and describes the language constructs for writing transactional programs. Section 2.2 discusses lower-level interfaces and the different designs choices for implementing the higher lever language abstractions. Because the focus of this dissertation is TM, the emphasis is more on optimistic and less on pessimistic design approaches. Section 2.4 describes the techniques for implementing TM in software and hardware. Again, because the work described in this dissertation is done using software transactional memory (STM) more emphasis is given to STMs. Section 2.5 summarizes this chapter.

2.1 Language Constructs

There are proposed several language extensions to support TM. Most of these extensions were proposed and refined in a set of specifications [4] after different feedbacks about using TM in real applications were provided by the research community as well as this work [42; 92; 100; 135]. This section describes a subset of the language extensions which were used in this work, mainly those proposed before or at the time when this work started. These language extensions are the `atomic` block, the `abort` and `retry` keywords, and the `tm_pure` and `tm_callable` function attributes. These

`atomic` blocks are used to denote a sequence of operations that should execute atomically. All operations which are inside `atomic` block are executed as a single atomic operation. For example, in Figure 2.1 both operations `statement1` and `statement2` will execute as one indivisible operation.

`abort` statement rolls back the execution of an `atomic` block back to the point before starting its execution and then resumes the program execution from the statement which follows immediately after the `atomic` block. For example, the `abort` in Figure 2.2 will roll-back the execution of the `atomic` (i.e. memory

```
atomic
{
    statement1;
    statement2;
}
```

Figure 2.1: Example `atomic` block.

```
statement1;
atomic
{
    statement2;
    statement3;
    abort;
}
statement4;
```

Figure 2.2: Using the `abort` keyword.

changes made by `statement2` and `statement3` and continue execution from statement4. Some implementations require that `abort` is within the static scope of an `atomic` block and others are more permissive and require that it is called in the dynamic scope of an `atomic` block. Such functionality is particularly useful for implementing failure atomicity. For instance, when an error happens, `abort` provides means to automatically restore the program execution to a safe point.

`retry` is used to coordinate the execution of `atomic` blocks [54; 97]. `retry` keyword is used to indicate a situation when the execution of an `atomic` block cannot proceed due to an unmet condition. In such a case, `retry` blocks the `atomic` block execution until an alternative execution path becomes possible. For example, in Figure 2.3 `retry` indicates that the execution of the `atomic` block cannot continue if the buffer is empty. In this case, the execution of the `atomic` block will block until a producer thread inserts an element into the buffer.

```
public Object Consume()
{
    Object x = null;
    atomic
    {
        if (buffer.IsEmpty) retry;
        x = buffer.GetElement();
    }
    return x;
}
```

Figure 2.3: Using the `retry` keyword.

`retry` can appear anywhere within the scope of `atomic` blocks. Unlike using explicit conditional variables in combination with `signal` and `wait` operations, `retry` does not require specifying the `atomic` block and the shared variables which are involved in the synchronization.

`tm_callable` is a function attribute which indicates that a function is called directly or indirectly from within an `atomic` block. This attribute is an explicit way to tell the compiler to generate a special transactional version for the functions declared as `tm_callable` and also to call the transactional versions of these functions from within the `atomic` block. Although, in most cases the compiler can statically deduce whether a function is `tm_callable`, static analysis is not sufficient for function pointers or virtual functions. If a function is not annotated properly the execution may have unexpected result. For example, in Figure 2.4 the `tm_callable` attribute is used to indicate that function `foo` is called inside an `atomic` block.

`tm_pure` is a function attribute which indicates that the function does not have any side effects. Example, `tm_pure` functions are the mathematical functions such as `sin`, `cos`, etc. which make a specific computation without modifying a shared state. This attribute can be used to give hints to the compiler to generate

```
tm_callable void foo();
...
atomic
{
    foo();
}
```

Figure 2.4: Using the `tm_callable` function attribute.

more optimized code for such functions. Also, this attribute can be used as a work around to obtain internal information about the TM state such as how many times an `atomic` block aborted. However, if it is used incorrectly, the TM system may not be able to properly roll back the aborted transactions and lead to unexpected program behavior.

2.2 Design and Operation

Atomic blocks are high-level language abstractions which the programmer can use for implementing concurrency relying on their well defined semantics. While the semantics of `atomic` blocks are expected to be precisely defined their low-level implementation is not so. Concrete implementations may follow different design approaches which depends on the taken assumptions and the runtime overheads. This section describes the operation and typical designs of transactional memory systems.

Atomic blocks can be implemented using other techniques such as lock inference. Therefore it is important to distinguish between `atomic` blocks as a language construct and their implementation whether TM or lock inference. With this abstraction the underlying implementation of `atomic` blocks can be changed with a more optimized one without affecting the programmers experience.

The operation of TM and lock inference is fundamentally different. TM follows an optimistic approach. It executes the `atomic` blocks speculatively and detects conflicts during their execution aborting one of the involved transactions. Conversely, lock inference follows a pessimistic approach. It acquires a set of

```
// Transaction Management
void StartTx();
void AbortTx();
bool CommitTx();

// Data access management
T ReadTx(T *addr);
void WriteTx(T *addr, T v);
```

Figure 2.5: A simple TM interface.

locks before executing a `atomic` block and releases these locks after executing the `atomic` block. The focus of this dissertation is TM and discussing more details about lock inference (which can be found at [27; 31; 48; 62; 80]) is beyond its scope.

2.2.1 Interface

A typical TM provides an interface for managing transactions and data accesses inside the transactions. Figure 2.5 shows a simple TM interface which can be supported both by STM and HTM. In this particular case, the TM interface is very much similar to the word based TL2 [33] STM library or the original HTM proposed by Herlihy and Moss [60].

The first set of operations in Figure 2.5 deals with managing transactions and the second set deals with managing the data accesses inside the transaction.

- `StartTx` is used to create a new transaction.
- `AbortTx` aborts the execution of the current transaction.
- `CommitTx` attempts to commit the current transaction. If the commit is successful, it returns `true` and if the commit is not successful it aborts the transaction and returns `false`.
- `ReadTx` is used inside a transaction and it returns the value at the address which is specified as a parameter.

```
// x = y = z = 0;
// Implicit
atomic
{
    x = 1;
    y = 2;
    z = x + y;
}
```

```
// Explicit - using TM interface
do
{
   StartTx();
   WriteTx(&x, 1);
   WriteTx(&y, 2);
   int x_v = ReadTx(&x);
   int y_v = ReadTx(&y);
   WriteTx(&z, x_v + y_v);
} while ( !CommitTx());
```

Figure 2.6: Example of using the explicit TM interface to implement an `atomic` block.

- `WriteTx` is again used inside a transaction and is used to update the value of an address which both are specified as parameters.

Figure 2.6 demonstrates with a simple example of manually implementing an `atomic` block with explicit calls to the TM interface, or how for example a compiler could automatically transform the `atomic` block into a lower level representation with explicit calls to the TM.

2.2.2 Data Versioning

To determine whether a transaction is valid TM systems track the memory references that are accessed inside an `atomic` through the `ReadTx` and `WriteTx` functions from the TM interface (see Figure 2.5 and Figure 2.6). Memory reads compose the *r*ead set and memory writes compose the *w*rite set. Typically there are two approaches how memory reads and writes are versioned. Read operations can be either *o*ptimistic or *p*essimistic. In optimistic reads, the TM system logs a version number associated with the referenced memory and validates the read set when the transaction attempts to commit. In pessimistic reads, the TM system locks the memory reference and detects conflicts when another transaction

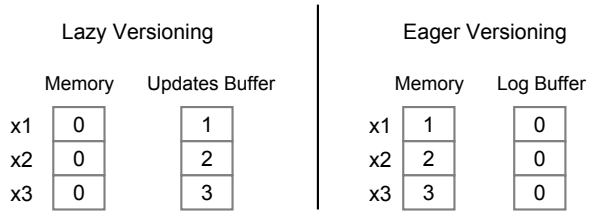


Figure 2.7: The state of the memory when using buffered updates and in-place updates just before committing the transactions from the example in Figure 2.6.

attempts to update the same memory location. Write operations can be *b*uffered (i.e. lazy versioning) or *i*n-place (i.e. eager versioning). With buffered writes the speculative values of the memory references are stored in a thread local buffer and when the transaction commits they are written back in their original locations to become visible to the other threads. With in-place writes, the TM system logs the original value for roll back in case an abort happens and writes the speculative value at its original place. Usually, in buffered update TMs, commits are more expensive whereas in in-place update TMs aborts are more expensive. Figure 2.7 shows how the memory would look depending on the type of the versioning just before committing the transactions from Figure 2.6.

2.2.3 Conflict Detection

In transactional applications a conflict occurs when two transactions access the same memory location concurrently and one of the accesses is write. TM may detect conflicts either *eagerly* (i.e. pessimistic) or *lazily* (i.e. optimistic). In eager approach conflicts are detected immediately when they happen whereas in lazy approach conflicts are detected at some later time of a transaction execution (e.g. commit).

Conflict detection can be different for the different conflict types – write-after-write (WaW), write-after-read (WaR) and read-after-write (RaW). For example, WaW conflicts can be detected eagerly and the WaR and RaW conflicts can be detected lazily. Table 2.1 illustrates the possible combinations.

Furthermore, TM implementation may differ based on the granularity at which conflicts are detected. Typically TM implementations detect conflicts at

WaW	WaR	RaW	Example
Lazy	Lazy	Lazy	TL2
Eager	Lazy	Lazy	TinySTM
Eager	Eager	Lazy	McRT
Eager	Lazy	Eager	n/a
Eager	Eager	Eager	LogTM

Table 2.1: Conflict detection can be different for the different conflict types. This table shows the possible combinations.

either word, cache line or object granularity. The choice of the granularity involves design and performance tradeoffs. Word and cache line granularity is more suitable for HTMs and non-garbage-collected lower level programming languages such as C whereas object conflict detection is more suitable for STMs managed object oriented programming languages such as Java and C#.

Detecting conflicts at machine word granularity requires more space for recording per-word metadata and also validation by iterating through the read and write set can be slower. On the other side, detecting conflicts at cache line granularity has lower space requirements and validation by iterating through the read and write set can be faster (because two or more words can map at the same cache line). However, TMs that operate at cache line may detect false conflicts when two transactions access different words of the same cache line. To mitigate the limitations of word based conflict detection, Riegel et al. [96] proposed a dynamic approach and Mannarswamy et al. [79] proposed a static approach for modifying the granularity of conflict detection for certain memory locations.

In TMs with object conflict detection a conflict occurs when two transactions access the fields of the same object and one of the accesses is write. Just like cache line granularity, this approach may signal false conflicts when different fields of the same object are accessed. In such case conflict detection at finer per-object field granularity is possible at the cost of managing more transactional metadata.

2.2.4 Conflict Resolution

Conflict resolution is a policy which determines how the TM system reacts when a conflict occurs. With respect to conflict resolution TMs may differ in two aspects – based on the time when a conflict is resolved and based on the way how the conflict is resolved. In time conflict resolution may follow immediately after a conflict is detected (i.e. *eager conflict resolution*) [87; 128] or it can be postponed to a later moment of the transaction execution (i.e. *lazy conflict resolution*) [24; 49; 119], for instance commit time. Once it is time to resolve the conflict the TM system may continue in one of the following ways:

- Blocks the transaction execution until the other conflicting transaction respectively commits or aborts. In case of a cycle with two or more waiting transactions the TM system aborts all of them to avoid deadlock [87; 128].
- Chooses one of the conflicting transactions as a victim transaction and aborts it. Scherer et al. [104] and Guerraoui et al. [104] studied different criteria of how to choose the victim transaction (i.e. the transaction to be aborted) and also the effect of delaying and assigning back-off time for the abort.

2.2.5 Commit

Commit happens at the end of the transaction after validation passes successfully (i.e. the transaction does not have any conflicts). The commit process makes the memory changes done during the transaction execution visible to the world (i.e. other threads). In eager versioning (i.e. in-pace update) TM systems commit has very low overhead, but in lazy versioning (i.e. buffered update) TM systems it might be expensive because the buffered updates should be written back to their original places in memory. For transactions with large write set lazy versioning approach may hurt the performance noticeably.

2.2.6 Abort

When a TM system detects a conflict it aborts the transaction by restarting its execution from the beginning. But before restart the TM system may perform

rollback to restore the program state to the point when it was before starting the transaction. In lazy versioning TM systems (i.e. buffered update) rollback is cheap but in eager versioning TM systems (i.e. in-place update) it may be expensive. When abort happens, lazy versioning systems only need to discard or clear the write set whereas eager versioning systems need to restore the original value for all speculatively updated memory locations.

2.3 Additional Functionality

This section describes functionality which can be provided by the underlying TM system to be used in different situations.

2.3.1 Nested Transactions

A nested transaction is one which execution is contained within the dynamic scope of another transaction. Depending on the implementation, nested transactions can be *flattened* or *closed*. These two types of nested transaction have the same semantics for commit. When a nested transaction commits successfully its read and write sets are merged respectively to the read and write sets of the outer transaction – in this case the changes made by the inner transaction become visible only to the outer transaction. However, the behavior of flattened and closed nested transactions is different when the nested transaction aborts. When a flattened nested transaction aborts it also aborts the outer transaction even if the outer transaction is valid. For example, in Figure 2.8 if Tx2 aborts, Tx1 will abort, too. On the other side, when a closed nested transaction aborts it does not abort the execution of the outer transaction. The changes made by the nested transactions become globally visible only if the most outer transaction commits successfully and otherwise they are discarded (see Figure 2.8).

Besides flattened and closed nested transactions there is also a third type of nested transactions – *open nested transactions* [6; 7; 88]. Unlike flattening and closed nesting, open nesting does not preserve the isolation semantics of transactions.

```
// x = y = 0;
atomic      // Tx1
{
    x = 1;
    atomic  // Tx2
    {
        y = 2;
    }
}
```

Figure 2.8: Nested transactions. *Flattening*: if Tx2 commits changes on `y` will be visible after Tx1 commits; if Tx2 aborts will cause Tx1 to abort as well; if Tx2 commits but Tx1 aborts changes made on both `x` and `y` will be discarded. *Closed*: if Tx2 commits changes on `y` will become visible after Tx1 commits (same as flattening); is Tx2 aborts will not cause Tx1 to abort (opposite to flattening); if Tx2 commits but Tx1 aborts the changes on both `x` and `y` will be discarded. *Open*: if Tx2 commits the value of `y` will be become globally visible even if Tx1 aborts (opposite to flattening and closed); is Tx2 aborts will not cause Tx1 to abort (same as closed).

The memory changes made by an open nested transaction become visible to the other threads immediately after it commits and they are not rolled back if the outer transaction aborts. For example, if Tx2 in Figure 2.8 is an open nested transaction and it commits successfully, the value of `y` will be 2 and it immediately will become visible to the other threads. Once Tx2 commits, the value of `y` will not be restored to 0 if Tx1 aborts. On abort, the behavior of open nested transactions is the same as closed nested transaction – only the nested transaction aborts without causing the outer transaction to abort. Although open nesting transactions are not compatible with the isolation semantics of `atomic` blocks they were proposed because of several practical reasons some of which are:

- To develop interactive transactional applications with user provided input;
- To optimize the performance of the application when the program algorithm allows to be relaxed; and
- To communicate transactional data which otherwise cannot be obtained, for instance, how many times a given `atomic` block has re-executed due to aborts.

2.3.2 Transaction Checkpointing

Transaction checkpoints are used delimit the transaction into rollback sections [123]. When conflict is detected, the transaction is rolled back until the point where the code before it is valid. The rationale of using checkpoints is to avoid rolling back and re-executing the valid part of the transaction. For example, if Figure 2.9 is shown a transaction which is checkpointed at two places – B and C. In this case, if a conflict is detected between B and C, the TM system will roll back and re-execute the B-D part of this transaction. On the other side, if no checkpoints are used, the complete transaction will be rolled back and re-executed.

A better technique that mitigates this limitation are abstract nested transactions (ANT) which are described in the next section.

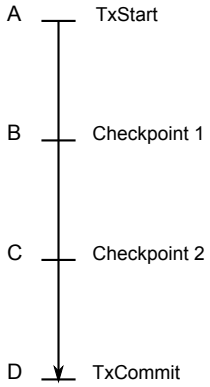


Figure 2.9: The transaction is checkpointed at points B and C. If a conflict is detected between points B and C, the TM system will roll back and re-execute only the B-D part of the transaction. If no checkpoints are used the TM system will roll back and re-execute the whole transaction causing more wasted work.

2.3.3 Abstract Nested Transactions

Harris and Stipic [53] have proposed abstract nested transactions (ANT) as a technique for optimizing the execution of `atomic` blocks which have benign conflicts. For example, if two transactions insert different items which keys map into the same bucket, the TM system will detect a conflict because both transactions update the same memory locations. However, in this case, the insert operation for is commutative and its concurrent execution is not a conflict at higher level of abstraction. ANTs are transparent to the enclosing `atomic` block and their semantics but they have different behavior. When ANT is not valid but the remaining part of the transaction is valid, only the ANT is re-executed. Operationally ANTs are similar to checkpoints except the extend of the code which is re-executed. To demonstrate the difference between checkpoints and ANTs let's assume that the B-C part of the transaction from Figure 2.9 is ANT. If at point D (i.e. when the transaction commits) the TM system detects that the ANT is not valid because of a conflicting memory access in B-C part, then then the TM system will roll back and re-execute only the B-C part. In contrast, when using checkpoints, in addition to the B-C part the TM system will also re-execute C-D part. This example show that ANTs can be more efficient than checkpoints because they would cause less code to be re-executed. Figure 2.10 shows what part

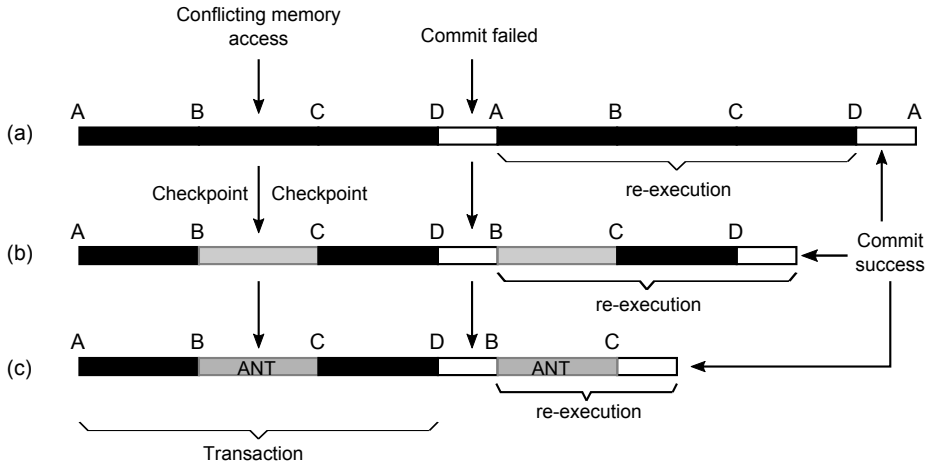


Figure 2.10: This figure shows the part of the transaction which will be re-executed when (a) a pure transaction is used, (b) the transaction is checkpointed at B and C, (c) the code between B and C is wrapped in ANT. It is assumed that the conflicting memory access is performed between the points B and C and the conflict is detected at point D when the transaction attempts to commit.

of the transactions will be re-executed when checkpoints and ANTs are used.

2.3.4 Transaction Scheduling

A TM system may support static or runtime transaction scheduling [35; 37; 78; 129]. Using this feature one can schedule two transactions to not execute at the same time. The rationale behind transaction scheduling is to reduce the abort rate of transactions by serializing their execution.

Typically in automatic transaction scheduling transactions are continuously monitored how frequently they abort. Whenever the abort rate exceeds a certain threshold transactions are serialized to reduce contention. Other approaches go step further by keeping history of the read and write sets of the transactions and try to predict weather two `atomic` blocks will conflict if they are executed concurrently. When possible the TM system may schedule two `atomic` blocks that are likely to conflict to execute on the same core. Unlike, dynamic scheduling, static scheduling cannot be flexible and adapt to the changing behavior of transactions. However, static scheduling does not have runtime overheads and might

```
// Initially x = 0;

// Thread 1          // Thread 2
atomic
{
    x = 1;           x = 2;
}
```

Figure 2.11: A TM with strong isolation will detect conflict between the transactional and non-transactional code. A TM with weak isolation will not detect conflict between the transactional and non-transactional code and will silently continue execution.

perform better in cases when the transactional characteristics of `atomic` blocks are constant. In addition, these two approaches can be combined to complement each others deficiencies – static scheduling can be used for the `atomic` blocks with predictive behavior and dynamic scheduling for those with non-predictive behavior.

2.3.5 Strong vs. Weak Isolation

Isolation is a property of transactions which ensures that the intermediate changes made during transaction execution are not visible until the transaction commits successfully. TMs with strong isolation guarantee isolation between transactional and non-transactional code and a TM with weak isolation guarantees isolation only between transactions [15]. Figure 2.11 shows an example of asymmetric race where variable x is assigned a value in a transactional code and also in a non-transactional code. A TM with strong isolation would detect such conflicts and a TM with weak isolation will these conflicts will pass undetected resulting in undetermined program execution.

Strong isolation eliminates a difficult to find set of race errors in multi-threaded programs thus making the semantics of transactions more intuitive. Then one would ask "Then why does the notion of weak isolation exists and not all TMs support strong isolation?". While strong isolation has no additional costs

in HTMs, it is not the case with STMs. Early, STMs with strong isolation incurred very high overheads because of instrumenting the non-transactional code with special calls to the STM in similar way how `atomic` blocks are instrumented (see Figure 2.6) [105]. Later on Harris et al. [1] have proposed more efficient implementation which uses the memory protection unit to detect asymmetric conflicts.

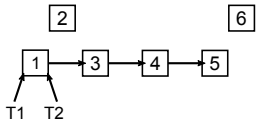
2.3.6 Handling Irrevocable Actions

Irrevocable operations such as I/O, system calls, etc. pose a challenge for transactions because it is not possible to roll back these operations when transactions aborts. There were proposed various techniques such as compensating actions [22; 51; 81; 131] or delayed output [72; 121] which try to address this problem. However, a general-purpose technique for handling irrevocable operations are irrevocable transactions [75; 116; 126]. Irrevocable execution is a fall-back technique in which a transaction is guaranteed to commit by not being involved in any conflict or winning all conflicts. Typically, when a transaction performs an irrevocable operation, it automatically switches to irrevocable mode.

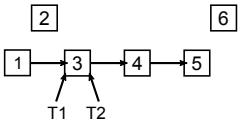
2.3.7 Early Release

Early release is a mechanism to exclude entries in the transaction's read set from conflict detection [41; 61; 109; 110]. In certain applications it is possible that the final result of an `atomic` block is still correct although it's read set is not valid. For example, consider an `atomic` block which inserts entries in a sorted linked list (Figure 2.12). Thread T1 wants to insert value 2 and thread T2 wants to insert value 6. To find the right place to insert the new values the two threads iterate over the the list nodes and consequently add them to the transaction's read set. T2 aborts because T1 finished faster and after inserting the new node it invalidates T2's read set. However, T2 could still correctly insert it's node although some entries in it's read set were invalid. In this case we can exclude all nodes except 5 from conflict detection.

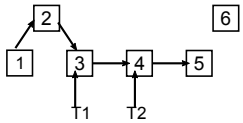
Although early release can improve application's performance significantly it is not a safe operation (i.e. early release can break program correctness). The



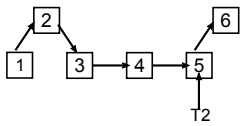
	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1		1	
T2	1		1	



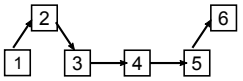
	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1, 3	1	1, 3	1
T2	1, 3		1, 3	



	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1, 3 COMMIT	1	1, 3 COMMIT	1
T2	1, 3, 4		3, 4	



	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1, 3 COMMIT	1	1, 3 COMMIT	1
T2	1, 3, 4	5	3, 4	5



	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1, 3 COMMIT	1	1, 3 COMMIT	1
T2	1, 3, 4 ABORT	5	3, 4 COMMIT	5

Figure 2.12: Transactions T1 inserts number 2 and transaction T2 inserts number 6 in sorted linked list. Without using early release T2 will abort and when using early release T2 will commit successfully.

programmer should precisely know the shared data structures and the operations applied on them. The available profiling tools can help in identifying the shared objects that are involved in conflicts and point which are the most critical ones. Provided with this information, the programmer can focus on the specific objects and try to use early release when possible or use different implementations for the data structures.

An alternative technique for early release are elastic transactions [40]. Unlike regular transactions which must execute to completion, an elastic transaction can make an intermediate commits when a conflict is detected. The intermediate commits are possible only before the transaction makes its first write. In effect, the intermediate commit is equivalent to releasing the read set of the transaction if the transaction is valid and its write set is empty. Compared to early release, elastic transactions provide cleaner support for composability.

2.4 Implementations

This section describes software transactional memory (STM) implementations and hardware transactional memory implementations (HTM). Typical STMs are implemented entirely in software as a runtime library. STMs are flexible, unbounded and does not require any change at the underlying computer architecture. Typical HTMs are implemented entirely in hardware as a micro-architectural extension of the CPU. HTMs have lower overheads than STMs but they are bounded both in space and time and also HTMs are not flexible.

2.4.1 Software Transactional Memory

A typical software transactional memory (STM) is implemented entirely in software as a runtime library. STMs provide a public API similar to the API in Figure 2.5. A compiler with STM support would automatically generate the code for the `atomic` blocks by calling the proper STM library functions as shown in Figure 2.6. In the absence of such compiler, the programmer should manually instrument the calls to the STM library. Manually instrumented code can be faster than the compiler generated code because the programmer may choose

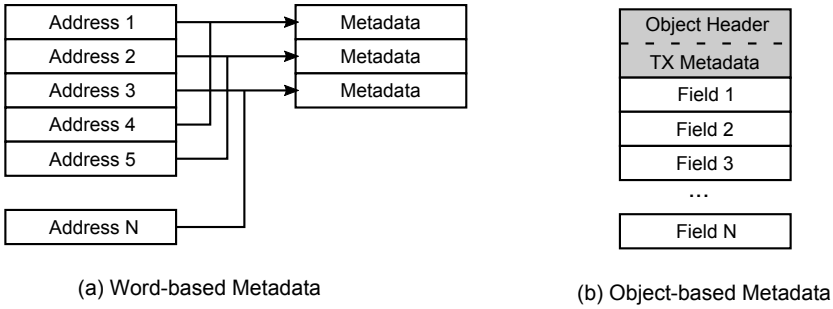


Figure 2.13: Transactional metadata: (a) word-based metadata, (b) object-based metadata.

to not instrument some of safe memory accesses such as thread local variables. However, a manual implementation can be very difficult and error prone because the programmer has to find all functions which are called inside transactions and also create transactional versions of these functions.

STMs represent the memory locations which are accessed transactionally through special transactional metadata. The STM metadata is either word-based or object-based. In word-based STMs [33; 96; 101] a memory location is mapped through address hashing to a transaction record in a fixed size metadata table (see Figure 2.13 (a)). In object-based STMs [5; 55] the transactional metadata is embedded in the object's header (see Figure 2.13 (b)). The information which is stored in the transactional metadata tells whether a memory location is read/written, its version number, state - locked/unlocked, the owner transaction, the original/speculative value, etc. Because of various trade-offs STMs organize and maintain the transactional metadata differently. This thesis includes description of only two STMs – TL2 as an example of word-based STM and Bartok-STM as an example of object-based STM. A more detailed description is beyond the scope this thesis and can be found at [56].

2.4.1.1 TL2

TL2 [33] is an example for a word-based, lazy versioning, lazy conflict detection STM. It is implemented in C and its programmer's interface is similar to the basic TM interface shown in Figure 2.5. A special object which represents a transaction maintains two lists of address – the read set and the write set (see

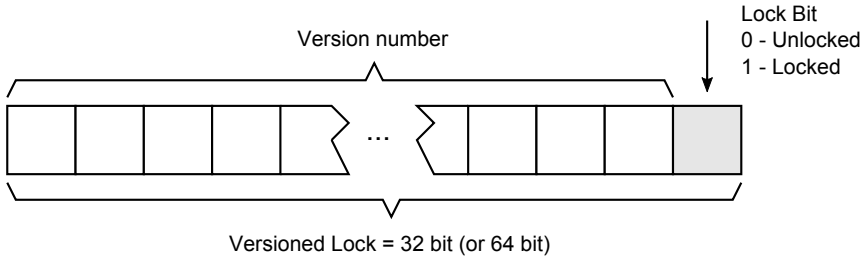


Figure 2.14: Versioned lock. The less significant bit indicates whether it is locked (if 1) or unlocked (if 0). The remaining part of the lock is a version number.

Figure 2.15). The entries of the read set are the memory addresses which are read by the transaction and their version numbers. The entries in the write set are the memory addresses updated by the transaction and also their speculative values. The records stored in the metadata table are versioned locks. Typically a versioned lock is word sized, its less significant bit indicates whether the lock is locked and the remaining part is a version number (see Figure 2.14).

The basic TL2 algorithm performs the following operations while it executes a transaction:

1. `TxStart` - samples the value of a global version clock into a *read version variable* (rv).

2. `TxRead` - checks whether the address is in the write set (i.e. it has been already updated by the transaction). If the address is in the write set, the speculative value is returned. If the value is not in the write set, the original value in the memory is read and the addresses together with its version number from the metadata is added to the read set. If the metadata entry is locked by another transaction or is greater than sampled version number during `TxStart` (rv) the transaction is aborted.

3. `TxWrite` - checks whether the address is already in the write set (i.e. it has been already updated by the transaction). If the address is in the write set, its speculative value is updated. If the address is not in the write set a new entry is created and added to the write set.

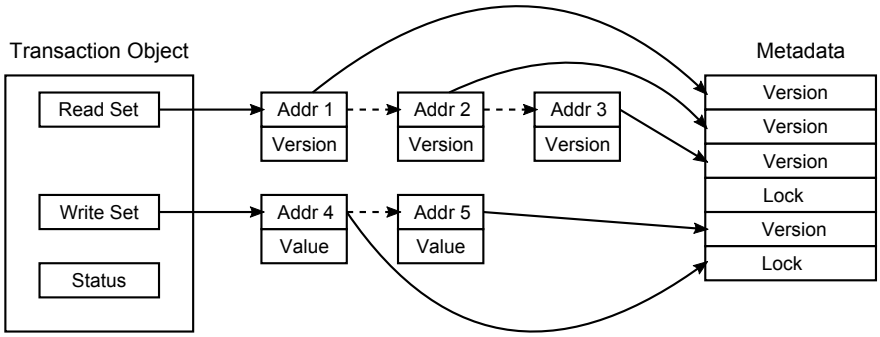


Figure 2.15: Managing the transactional metadata in TL2

4. `TxCommit` - the commit operation proceeds in three phases. During the first phase (*lock write set*) the write set is locked using two-phase locking; the metadata entry associated with every write set entry is locked. If the locking does not succeed the transaction aborts. If the locking succeeds the global version clock is sampled into a *write version variable* (*wv*) and the execution proceeds to the second phase. During the second phase (*read set validation*) TL2 validates the read set of the transaction. For successful validation is necessary that the version of every entry in the read set is less than or equal to the version number sampled at `TxStart` ($version \leq rv$). If the version of any read set entry is greater than the version at `TxStart` ($version$ ¿ rv) the transaction is not valid and it is aborted. If the validation succeeds the execution proceeds to the third phase. During the third phase (*write back and release*) the speculative value of every write set entry is written to the memory and the metadata associated with this entry is unlocked by writing the *write version* (*wv*) to its place.

More efficient versions of TL2 use thread local version clocks to reduce the contention on the global version clock.

2.4.1.2 Bartok-STM

Bartok-STM [55] is an example for an object-based, eager versioning, lazy conflict detection STM and most of the work described in this thesis was build on top of Bartok-STM. Bartok-STM is implemented in C# and is part of the Bartok

compiler. Bartok compiler is an ahead of time C# to x86 compiler with language level support of `atomic` blocks and transactional memory.

In Bartok-STM the TM metadata is embedded into the object header. The object header is word sized and it might be used for several different purposes. The type of the use is determined by the value of the first two bits (see Figure 2.16). Initially all the bits in the object header are zero indicating that the it has not been used. If the value of the first two bits is:

1. 00 – the remaining part of the word encode a version number;
2. 01 – the object is locked for write by a transaction and the remaining bits (including the second bit) hold a pointer to the owning transaction;
3. 10 – the header is used for different purpose other than transaction management and the remaining part of the bits hold the hash code of the object.
4. 11 – the object's header is used for more than one of the above purposes and the object header is inflated and stored in an external structure. The remaining bits of the object header are a pointer to the inflated structure, which holds the object's version number, owning transaction and hashcode.

Each thread which is executing a transaction is represented with through a special object – transaction manager (see Figure 2.16). Unlike TL2, the transaction manager maintains three lists – read set, write set and undo log. Each entry in the read set has two fields – reference to the object opened for read and the version number of the object at the moment when it is accessed. Each entry in the write set stores a reference to the object opened for write. Each entry in the undo log has three fields – reference to the object opened for write, the offset of the modified field and the original value of the field.

The programmer's interface of Bartok-STM is slightly different than the basic TM interface from Figure 2.5 and is shown in Figure 2.17 (McRT-STM [101] has similar interface).

Essentially `StartTx`, `AbortTx`, and `CommitTx` are the same as in the simple TM interface from Figure 2.5 (see Section 2.2.1). `StartTx` starts a new transaction, `AbortTx` aborts a transaction, and `CommitTx` attempts to commit a transaction.

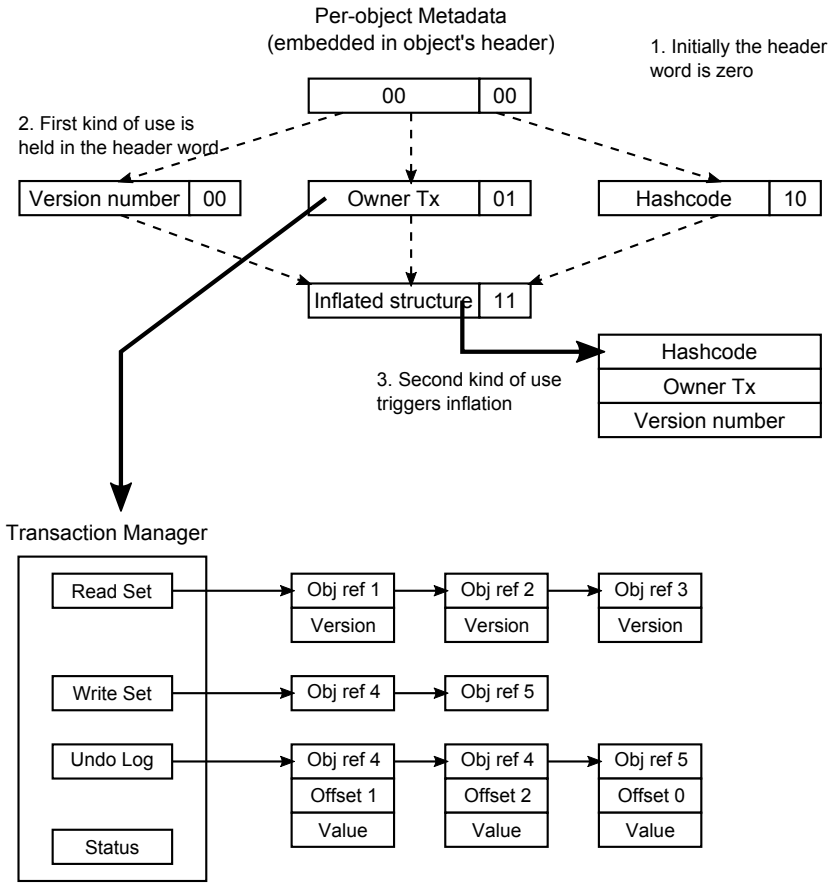


Figure 2.16: Bartok-STM stores the TM metadata in the object's header. It organizes the metadata using three lists – read set, write set and undo log.

```
// Bartok Interface
void StartTx();
void AbortTx();
bool CommitTx();

// Data access management
void OpenObjectForRead(Object obj); void OpenObjectForWrite(Object obj);
void LogObjectFieldForUndo(Object obj, int fieldOffest);
```

Figure 2.17: The programmer's interface of Bartok STM.

```
  // Non-instrumented
1: atomic {
2:    local.field1 = x.field1;
3:    y.field2 = 5;
4: }
5:
5:
6:
      (a)
```

```
// Instrumented code
atomic {
   OpenObjectForRead(x);
   local.field1 = x.field1;
   OpenObjectForWrite(y);
   LogObjectFieldForUndo(y, 2);
   y.field2 = 5;
 }
        (b)
```

Figure 2.18: Using the interface of Bartok-STM. (a) non-instrumented `atomic` block, (b) `atomic` block instrumented with calls to `OpenObjectForRead` before the actual read operation, and `OpenObjectForWrite` and `LogObjectFieldForUndo` before the actual write operation.

In Bartok-STM `ReadTx` can be expressed through `OpenObjectForRead`. `OpenObjectForRead` must be called for every object before the actual reading operation (see Figure 2.18). `OpenObjectForRead` logs the address and the version number associated with the object into a read set entry.

In Bartok-STM `WriteTx` operation is split into two operations `OpenObjectForWrtie` and `LogObjectFieldForUndo`. Both methods, `OpenObjectForWrtie` and `LogObjectFieldForUnd` must be called before the actual write operation (see Figure 2.18). If the object is not opened for write by another transaction `OpenObjectForWrite` locks the object by writing a pointer to the thread's transaction manager and adds an entry to the write set. However, if the object is opened for write by another thread (i.e. the object is locked and the value at the metadata is a pointer to another transaction manager) the transaction aborts. `LogObjectFieldForUndo` logs the original value of the object's field into an undo log. In case transaction aborts, the original values of the object's fields are resorted from the undo log.

Transaction's read set is validated before committing the transaction. During the validation, Bartok-STM checks whether the version number of the read set entries has changed by comparing it with the version number stored at the actual object. If the version number is different or the object is opened for write by another thread (transaction manager) validation fails and the transaction aborts. If the validation is successful, Bartok-STM unlocks the objects in the transaction's write set by incrementing their version numbers and writing them to the objects

metadata. When transaction aborts, Bartok-STM walks through the undo log and restores the original value of speculatively updated objects.

2.4.2 Hardware Transactional Memory

Transactional memory systems can be implemented entirely in hardware – hardware transactional memory (HTM). Like STMs, HTMs perform the same set of transactional operations. However their programmer interface may differ as explicit or implicit. An explicit interface extends the ISA with additional memory instruction for transactional load and store, similar to the STM's interface in Figure 2.17. An implicit HTM interface has instructions indicating the start and the end of the transaction. Such TMs implicitly treat all memory operations as transactional.

As with STMs, when a transaction starts the HTM saves the CPU registers to be able to restore the state in case a conflict happens. Unlike STMs, HTMs track the transactional memory accesses into the CPU caches by extending their cache tags. Lazy versioning HTMs [24; 49] buffer memory updates in the local caches and broadcast to the other CPUs on successful commit. Eager versioning HTMs [87] log the original value of the memory for roll-back in case the transaction aborts and broadcast the memory update. To prevent other CPUs from consuming speculative updates, the HTM blocks every CPUs which attempts to access a speculative memory by not-acknowledging its request. Because of possibility of deadlock, HTMs use mechanisms to detect cycles when CPUs are blocked.

HTMs detect conflicts automatically by leveraging the underlying cache coherence protocol or using supporting structures such as bloom filters [128]. Aborts in lazy versioning HTMs are trivial and require only local updates on the CPU cache tags to restore the CPU state to the point before starting the transaction. Aborts in eager versioning broadcast the transaction's write set to acknowledge the CPUs which might be waiting for accessing a memory on its read set.

More detailed discussion about different HTMs can be found at the following references [24; 49; 60; 87; 107; 119; 128].

2.4.3 Hybrid Transactional Memory

Hybrid transactional memory system have both software and hardware component. The goal of proposed hybrid TMs is to mitigate the limitations of both STMs and HTMs. Typically, STMs are accelerated by adding a hardware support [57; 86; 102; 106] and HTMs are virtualized by adding software support [8; 28; 32; 68; 94]. Like STMs and HTMs, hybrid TMs perform the same set of transactional operations described earlier in Section 2.2. There are many hybrid TM proposal which differ in their implementation. Detailed discussion for each implementation is outside the scope of this dissertation but can be found at the following references [8; 28; 32; 57; 68; 86; 94; 102; 106].

2.5 Summary

This chapter have introduced `atomic` blocks and how they are implemented with transactional memory. `atomic blocks` are language level constructs. A code wrapped inside an `atomic` block executes atomically as if it is a single operations. Transactional memory is an optimistic concurrency control which executes `atomic` blocks atomically and in isolation. A typical TM provides an interface for managing transactions and data versioning. A compiler with TM support uses this interface to instrument the body of `atomic` blocks or alternatively this can be manually done by the programmer. The design of TM systems involve various performance and implementation trade-offs. TMs can be implemented either in software or in hardware. STMs are flexible but slow. HTMs are fast but are not flexible. There are hybrid approaches which try to either accelerate STM with hardware support or make HTMs unbounded and more flexible with software support.

Chapter 3

Developing Programs with Atomic Blocks and Transactional Memory

Transactional memory (TM) as a promising alternative to locks is actively being studied by the research community. Much of the initial work on TM used microbenchmarks and application kernels to evaluate the performance of TM implementations. These workloads are small and usually stress specific aspects of the TM implementation such as the commit or abort operation. While they are useful in evaluating and tuning the performance of internal TM structures it is not clear whether conclusions drawn from these workloads will apply to large real applications. Also, these applications are often developed by TM researchers and they do not necessarily reflect how TM would be used by programmers who are not aware of how TM is implemented.

This chapter describes the TM applications which were developed as a part of this dissertation. Section 3.1 follows with the motivation of developing these applications. Section 3.2 surveys the existing TM applications and relates them with our work. Section 3.3 describes an experience of using `atomic` blocks and TM in a real world parallel application – Quake game server. Section 3.4 describes a synthetic transactional memory workload which can be configured to stress specific internal TM structures or model the transactional behavior of a real TM

application. Section 3.5 describes the implications of porting applications from the STAMP benchmark suite from C to C# programming language.

3.1 Motivation

Several transactional memory applications were developed as a part of this dissertation. Each of these applications was motivated because of different reasons.

AtomicQuake AtomicQuake [135] is a large real-world application derived from a parallel version of the Quake game server [2]. The motivation of developing AtomicQuake was to:

1. see hot TM is used in a setting of a large real application and to verify whether indeed developing parallel programs by TM is easier than locks;
2. see whether the performance of TM is comparable to the performance of locks;
3. see whether TM is mature to develop production software (i.e. well defined semantics, language and tool integration, interoperability with locks, etc.);
4. deliver the research community a new real workload to test and benchmark the complete TM implementation and integration.

WormBench WormBench [134] is a configurable synthetic workload for TM. The motivation of developing WormBench was to have a workload that can be easily configured to:

1. reproduce specific pathological executions which stress specific weak or strong designs and implementations of TM; and
2. model the transactional characteristics of existing workloads.

STAMP Several applications from the STAMP benchmark suite were ported from C to C#. These applications were used for evaluation and analysis of Bartok-STM extensions. Our choice of using STAMP but not other application, for example AtomicQuake, is that

1. STAMP applications have small code base (about 1000 LOC) which made it easier and faster to port; and

2. each STAMP application is a representative of a real applications because it implements an algorithm which is used in large applications.

On the other side, considering that development of AtomicQuake took more than 12 months porting it from C to C# would take quite significant time.

3.2 Related Work

Early work used simple data-structure microbenchmarks such as linked-lists, red-black trees, and skip-lists to evaluate TM implementation [29; 60]. These applications perform simple lookup, insertion and deletion operations without doing real work. These applications are good for evaluating the TM system's implementation details such as the size of the internal data structures or cashes. However, they are not representative for evaluating the overall TM system in a setting of a real application which does real work with while operating on the basic structures.

STMBench7 [46] is derived from the OO7 benchmark [21] for object-oriented databases. STMBench7 performs long-running operations which update and traverse a complex graph-based data structure. Because there is no real purpose of performing these operations it can be classified as a synthetic workload. There exists Java and C++ implementations of STMBench7 with medium-grained and coarse-grained synchronization. The Java implementation uses annotations to identify transactions. The C++ version uses explicit function calls to access an STM library. Transactions in STMBench7 contain recursive calls. Transactions in STMBench7 do not have I/O operations, system calls, exception handling or privatization patterns. Compared to the other workloads even the shortest

transactions are of 100x magnitude larger than the transactions in other applications. This makes it suitable for evaluating STMs limits [36] and also HTM-virtualization techniques.

TMunit [50] is an extensive framework for testing and evaluating STM libraries. It provides a domain specific language for writing unit tests. These tests can specify particular interleavings of threads in order to recreate problematic scenarios – e.g. when there is a non-transactional access to a memory location that occurs concurrently with a transactional access to the same location. TMunit can also generate test workloads by analyzing traces from the lock-based execution of a parallel program. Transactional workloads generated by TMunit run on top of an interpreter that makes direct calls to an STM library.

EigenBench [64] is a small synthetic microbenchmak which is similar to WormBench but implemented in C. Just like WormBench, EigenBench can be configured to have different transactional characteristics such as read/write set size, abort rate etc. In this way, one can recreate pathological execution which in normal situations happen rarely or also model the transactional characteristics of real applications. Unlike WormBench, EigenBench does not perform interleaving dummy computations.

Kang and Baded [66] developed a parallel algorithm using STM which finds a minimum spanning tree in a graph. They report that using TM is easy to implement fine-grained synchronization for a complex data structures. However, in their experiments the overall performance is not satisfiable due to the STM overheads. In a later paper Dice *et al.* [34] demonstrated how using the HTM extensions of the Rock processor can deliver better results than locks on the same minimum spanning tree algorithm. They improved the base minimum spanning algorithm by privatizing large structures inside a transaction and then operating on the object non-transactionally.

LeeTM [125] is a parallel version of the Lee's path routing algorithm. LeeTM uses TM to implement the synchronized access to the underlying matrix which represents the circuit. Later on, STAMP TM benchmark applications suite has included a similar program with the name Labyrinth.

STAMP [20] is a suite of applications written to benchmark different TM implementations. The independent applications in the suite are algorithm kernels

with different characteristics in terms of how long they spend running inside transactions, how large those transactions are, and how likely concurrent transactions are to conflict with one another. The STAMP applications can be configured for use with HTM (in which only the start and end of each transaction is identified in the source code), or for use with STM (in which case the shared memory accesses are also made explicit). The STM configuration therefore models the behavior of a compiler that can avoid the use of STM on memory accesses to thread-local locations. The structure of the atomic sections is simple – without nested transactions, privatization patterns, system calls, I/O, and error handling. STAMP does not have lock-based implementations of the applications, although the behavior of variants using a single global lock, in place of transactions, has sometimes been studied [1].

Haskell STM Benchmark suite [93] is a collection of programs, ranging from small synthetic workloads (e.g. contended access to a shared counter), to applications written by programmers who were not STM researchers (e.g. a parallel solver for Su Doku puzzles). Although the Haskell STM API is similar to library-based STM APIs in imperative languages, the core functional parts of these benchmarks are very different from current mainstream languages. It would be difficult to rewrite them in a language like C# or Java.

SPLASH-2 [127] is a benchmark suite of highly parallel applications which have subsequently been adapted to use TM for synchronization [89]. In general, the SPLASH-2 applications involve short, infrequent, critical sections. These make up a small proportion of the overall execution time and preferred for HTM evaluation.

RMS-TM [67] consists of several applications derived from existing recognition, mining and synthesis workloads and are implemented in C and C++. Execution of RMS-TM applications exhibit high degree of parallelism and spend very little time in transactions. Typically they have few short critical sections which access few shared variables. Because of its transactional characteristics, this benchmark is an example showing when TM performs as good or even better than locks.

QuakeTM [42] is a transactional memory version the Quake game server [65]. Both QuakeTM and AtomicQuake were developed concurrently as part of the

same project but had different goals. The approach in QuakeTM was to study how TM applications can be developed from the scratch and the goal of AtomicQuake was to study how existing lock-based applications can be adapted to use TM as a basic synchronization (see Section 3.1). Because QuakeTM was developed from the scratch and with knowledge about `atomic` blocks and TM, it reported about fewer problems which were relevant to the use of `atomic` blocks. For example, in QuakeTM there were no instances of restructured the code with the sole purpose to match the block structure of `atomic` blocks. However, other problems such as library calls (e.g. `sprintf`) were common for both AtomicQuake and QuakeTM. Also both, QuakeTM and AtomicQuake made similar conclusions with respect to STM's performance that STM's overhead is high and should be reduced.

SynQuake [77] is another transactional version of the Quake game server. Unlike AtomicQuake and QuakeTM, SynQuake is a stripped version which includes only the main data structures and the essential features of Quake and excludes other secondary elements such as 3D space, network communication, etc. Opposite to the findings in AtomicQuake and QuakeTM (high abort rate and STM overhead) SynQuake showed performance which is competitive to a lock-based version of the same game engine. The performance improvement is achieved through an important TM and game specific optimization and also STM extensions which are tailored to the game's logic. To reduce the conflicts SynQuake implements dynamic locality-aware assignment of tasks to threads. Because of the lower conflict rate SynQuake shows better scalability and performance over AtomicQuake and QuakeTM.

Transactional Space Wars 3D [11] uses TM to implement the synchronization in a parallel version of the game. The findings of this work are aligned with the conclusions made in AtomicQuake and QuakeTM: STM is easy to use but has high overhead. To improve the performance of the game, authors use privatization to access the local copies of the shared data outside transactions. However, this approach complicates the clean basic implementation because additional merge functions are introduced between the states of objects that have experienced a conflict

TxLinux [95; 98; 99] is a variant of Linux operating system that uses hardware transactional memory (HTM) as a synchronization primitive. TxLinux imple-

ments a special-purpose co-operative transactional spinlock (`cxspinlock`) which allows mixing locks and transactions. `cxspinlock` allow locks and transactions to protect the same data while maintaining the advantages of both synchronization primitives. Initially the system attempts the execution of the critical section using a transaction. If the execution encounters a non-revocable operations such as an IO operation it falls back to lock. Transactional use of the `cxspinlock` is prevented while any thread holds it as an actual lock; this prevents problems with mixed transactional and non-transactional accesses.

3.3 Atomic Quake

This section discusses an experience of building *AtomicQuake*. AtomicQuake is a multi-player game server derived from a parallel lock-based version [2] by replacing all the lock-based syncretization with TM. There are several reasons which motivated the development of AtomicQuake and these are:

1. understand the TM programming idioms in the setting of a large application and verify whether TM is indeed easier to use than locks. The aim of studying a large, real, application was to gain insights about many of the choices that researchers are considering when designing programming abstractions based on TM. For example, whether or not strong atomicity is required [1; 12; 14; 82; 105], whether TM is useful for failure atomicity as well as synchronization, how frequently open nesting [88] or transactional boosting [58] are useful, which kinds of library calls, system calls, or IO are used in transactions [16].

2. compare the performance of TM and locks in the setting of a large and real application which exercises not only the lower level structures used in the TM implementation but also the complete TM implementation such as compiler integration and optimizations. For example, in most TM workloads, `atomic` blocks are manually instrumented with calls to the TM API. This is an idealistic approach which assumes the availability of a perfect compiler which can distinguish all the safe memory operations from the non-safe ones.

3. see whether TM is a mature technology which can be used to develop production stable software. For example, are semantics of TM are well defined for the corner case (i.e. how exceptions are handled inside transactions), are language level extensions for TM are expressive, do existing software development tools such as compilers, debuggers and profilers support TM and how they can be extended to become TM aware.

4. deliver the research community a new tougher workload to test and benchmark the complete TM implementation together with its system integration. The `atomic` blocks in most TM workloads are manually instrumented with calls to the STM library or HTM API. Their transactions has regular are simple (e.g. without nesting, library calls, IO etc.) and has regular runtime characteristics. Therefore such applications cannot exercise the complete TM implementation and intergradation.

The experience of using TM in AtomicQuake showed that indeed parallel programming with TM is easier than locks. TM makes fine-grained synchronization of many object and also complex data-structures such as tree trivial (Section. However, in AtomicQuake it was not easy to replace all lock-based synchronization with `atomic` blocks. The main difficulty stemmed from the different approach of using locks and `atomic` blocks: locks are lower-level synchronization abstraction used to manually implement atomicity on the shared data structures whereas `atomic` blocks provide atomicity transparently. Therefore it was necessary to reverse engineer the association between locks and the data data structures which they protect. An additional challenge was understanding the locking protocols used to acquire locks in specific order to avoid deadlock.

When compared with locks, the STM version of AtomicQuake scaled well but STM had prohibitively high overhead – about x5 times on a single threaded execution and more on multi-threaded execution (see Figure 3.17). Obtained results were contradictory with many other existing evaluations which used smaller workloads and also a later research which developed and evaluated a different version of Quake – SynQuake [77]. As discussed earlier in the text, one reason which explains the mismatch in the results is that most of the workloads used to

evaluate TMs are small and their `atomic` blocks are manually instrumented, including SynQuake. This implicitly assumes the perfect compiler that can exactly tell the memory accesses to shared data and instrument only these. For example, Yoo *et al.* [130] showed that the performance of the STAMP applications is lower when the `atomic` blocks in these applications are automatically compiled with a prototype STM enabled C.

During the AtomicQuake development there were encountered many problems it was either partially supported or even not supported by the available software development tools. One part of the problems were related to the semantics of TM. For example, it was not clear how errors are handled and recovered inside transactions. Other part of the problems were related to the TM language extensions. For example, many times the use of `lock` and `unlock` operations did not match the block structure (i.e. how `atomic` blocks are used) and it was necessary to restructure the code. At the beginning, the compiler failed to to compile the code and it was necessary to find ad hoc foregrounds. It was extremely difficult to debug and profile the code because existing debuggers and profilers were not aware of `atomic` blocks and TM.

The development of AtomicQuake took about 12 man months. The resulting implementation comprises 27 400 lines of C code in 56 files. On a fully loaded server, about 98% of the request processing part in request processing (RP) phase (Figure 3.2) executes in transactions, and the RP part as a whole constitutes about 63% of the total execution time. AtomicQuaje has 61 `atomic` blocks. Some of these `atomic` blocks are on the critical path and others not. Almost all `atomic` blocks contain function calls. On average the static call graph for a typical `atomic` block is 4 levels deep and contain 20-25 functions. Inside the `atomic` blocks there are IO operations and system calls. There are long and short running transactions (200–1.3M cycles) with small and large read and write sets (a few bytes to 1.5MB). There are nested transactions reaching up to 9 levels at runtime. There are examples where error handling and recovery occurs inside transactions. There are also examples where data changes between being accessed transactionally and accessed non-transactionally. Transactions in AtomicQuake has diverse runtime characteristics some of which also change during program execution. Compared to other existing TM workloads, AtomicQuake is a very

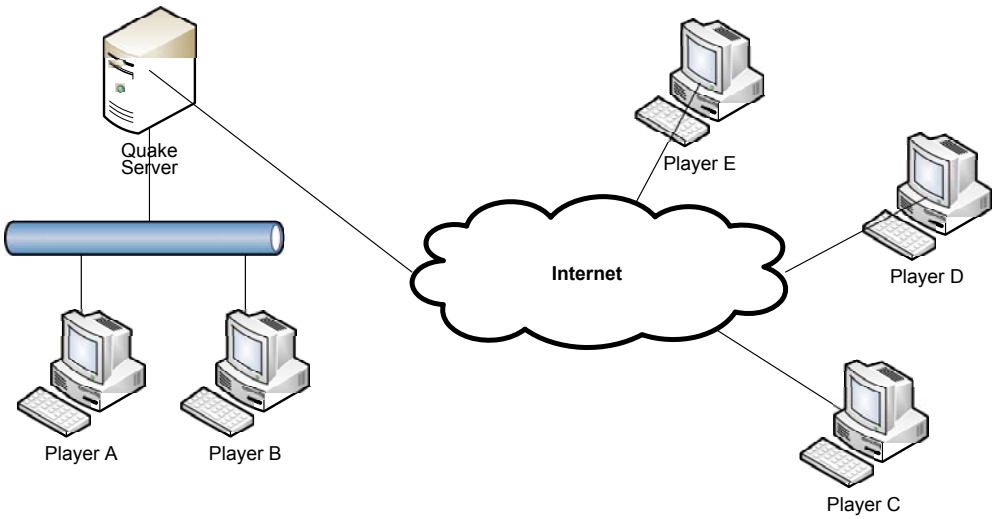


Figure 3.1: The client-server model in a multi-player Quake game session.

rich TM workload which exercises wide range of of the design and implementation aspects of a complete TM system. These features make it though for many TMs to compile and execute AtomicQuake and probably this is the reason why no other work to date used AtomicQuake as a benchmark.

The remaining part of the section continues with a detailed description and discussion on the experience of using `atomic` blocks and TM in AtomicQuake. Section 3.3.1 makes an overview of the Quake game server and describes its parallelization approach, shared data structures, and the synchronization between the threads which operate on these shared data structures. Section 3.3.2 describes how lock-based synchronization was replaced with `atomic` blocks and the experience of this process. Section 3.3.3 discusses the performance and the runtime characteristics of AtomicQuake.

3.3.1 Quake Overview

This section discusses the existing parallel implementation [2] of the Quake game server [65] on which AtomicQuake derives from.

Quake is an interactive first shooter game that renders the game world from the visual perspective of the player character. The multiplayer version of Quake, henceforth referred as Quake, is based on the client-server model (see Figure 3.1).

Players connect to the server and interact among each other by sending requests to the server to convey their intended actions. The server is a hub that maintains the game plot and drives the interactions between the players. The client component of the game renders the graphics and sound based on the individual messages sent by the server. The requests that clients send to the server are of two types: game session management (e.g. connect, disconnect) and interaction with the game world (e.g. move, shoot). For our research work, we are mainly focus on the second type of messages since they have the most significant impact on the runtime/execution of the server. To maintain a consistent game world for all players, the server maps the effect of players' actions on global state. In the parallel version of the Quake described in Section 3.3.1.1, accesses to the date structures storing the game state is guarded by locks to avoid races. In Section 3.3.1.2 we describe the shared data structures such as the game state and entities, that worker threads repeatedly update to reflect the results of processing the client requests and how these shared data structures are synchronized with locks.

3.3.1.1 Parallel Quake

The implementation of the parallel Quake [2] is based on the shared memory model and done with the `pthread` library. The execution in the server side is decomposed into three distinct phases that are synchronized with global barriers: Update world physics (U), Receive and Process client requests (RP) and Send replies to clients (S). The (RP) and (S) phases are executed in parallel by multiple threads, whereas (U) phase is not parallel and executed by one thread.

Figure 3.2 shows the game cycle in the parallel Quake with the execution breakdown in the following phases measured with one thread when the server is fully loaded [3]. Every server thread spins in the main loop waiting for client requests. Each iteration over the loop creates a new *frame* that can be thought of as a discrete representation of the game state on a certain time slice. The first thread that receives client request is identified as the *master thread*. The master thread first updates the physics (U). Updating the physics takes insignificant time with respect to the processing of the frame and is not parallel. If other

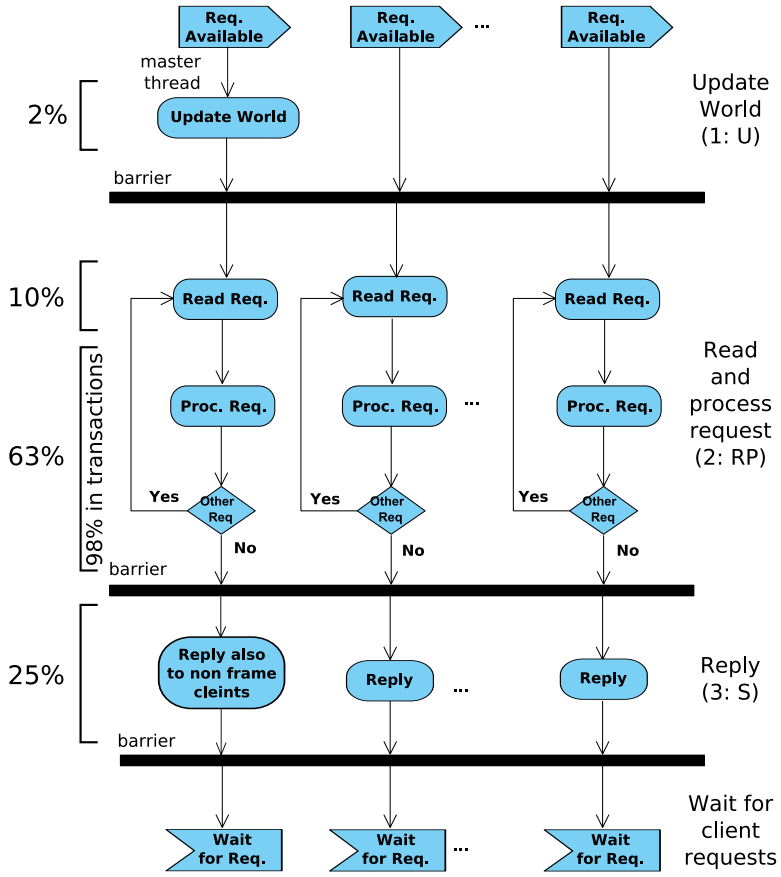


Figure 3.2: The game cycle in the parallel Quake server with the execution time in the different phases (only the threads that have received a request are shown).

threads receive a client request while the master thread is updating the physics they wait for the master thread to finish (U) and to signal the start of the new frame. When the start of the new frame is signaled by the master thread, all the waiting threads wake up and enter in (RP) phase. These threads are identified as *frame threads* because they participate in preparing the new frame. Preparation of the new frame is parallel and every frame thread executes this part of the code independently. When preparing the new frame, each thread first reads the client requests. Clients are statically assigned to the worker threads. These clients that send request to the server are said *to participate in the frame* and are named as *frame clients*. Every client request is processed independently in the `SV_ExecuteClientMessage` function which incrementally builds the new frame as a reply message. After a thread processes all the client requests, it waits for the other threads's finish and the frame completion. When all the threads are done with the client requests they enter into send requests phase (S). Threads send reply messages only to the frame clients (i.e. the clients that the thread received a request from) and all the clients that did not take action in the current frame are updated with reply messages which are sent by the master thread. At the end of (S) phase, threads wait until all the reply messages are sent and start again over.

Threads that receive a client request while a frame is being processed are said *not to participate in the frame*. These threads wait until the current frame completion (i.e. all the reply messages are sent) - the end of current frame is signalled. Then the first thread that wakes up (if there are any that missed the frame) becomes the master thread and starts doing physics (U) and everything else continues as described above.

3.3.1.2 Shared Data Structures

The parallel Quake server utilizes three different data structures that can be accessed concurrently. These data structures are: per-player reply message buffer, a common global state buffer and game objects.

All pending messages that has to be sent to the client are first accumulated in the client's reply message buffer. The source of these reply messages are other

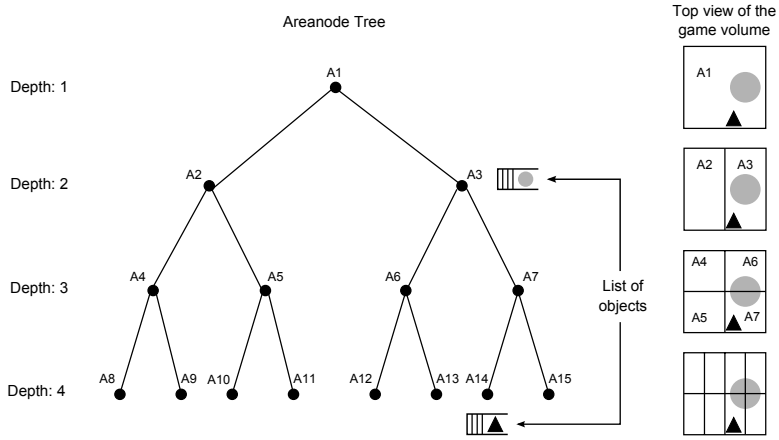


Figure 3.3: Mapping game volume into areanode tree. The figure has been adapted from [2].

other clients. An example reply message is a text message between players or a server message which informs when a player joins or leaves the game session. Because multiple threads can write client's message buffer, the thread access to every message buffer is synchronized with locks, one per buffer.

The global state buffer is updated during the world physics update phase (U) and clients' request processing phase (RP). This buffer is used to update all clients, regardless of whether or not the server received a request from a client during the current frame: each thread participating in the current frame uses this buffer to update the message buffers of its complete set of clients and the master thread performs this operation for clients belonging to threads not participating in the current frame. During the world update phase the global state is accessed by one thread only and there is no necessity for concurrency control. During request processing phase, multiple threads may attempt to update the global state buffer at the same time. Therefore to avoid races the global state buffer is guarded with a single lock.

When a player executes an action, it interacts with the other players and entities on the game map. These game entities are referred to as game objects. Before executing an action, the server identifies the list of the objects on the map that the player is likely to interact with and locks them to avoid concurrent access.

The server uses a fast-path for locating the objects in the map. The fast-path is achieved through a balanced binary tree data structure with depth 5 (including the root node). The three data structure is called *areanode tree* and its nodes are called *areanodes*. Each areanode represents a part from the entire 3D volume of the game. The children of an areanode represent two equally sized volumes that are obtained by dividing the volume represented by their parent with a vertical plane on either X or Y axis. The division of the entire game volume into more refined smaller areanodes is done recursively. At each level the selection of the division axis alternates between X and Y. All the areanodes in the same depth are a refined representation of the entire game volume. Moreover, each areanode has a list of the objects that are located in the volume that it represents. If an object falls in two areanodes (volumes) after division, then it is not associated with either of them but with their parent. This is better explained with an example areanode tree shown in Figure 3.3.

In Figure 3.3, on the left is the areanode tree that abstracts the division of the game volume which is on the right (the game volume on the right is a top-view). The root node of the tree is labeled *A1* and represents the entire 3D volume in the game. Its children, *A2* and *A3*, represent the two equally sized volumes obtained after partitioning the entire game volume with a vertical division plane on the Y axis. The triangle object is not crossing any division plane and it is in the list of objects associated with the *A14* leaf only. The circle object intersects with the plane that divides *A3* therefore it is put in the list of objects associated with *A3*, but not *A6* or *A7*.

The serialized access to the game objects is done by region-based locking scheme [70; 118]. To ensure an exclusive access to the objects that a player interacts with, threads lock regions of the map prior to processing a request. This is achieved as a consequence of locking the leaves in the areanode tree that contains an object the player would interact with (note that this results to a locking of objects that the player does not interact). If an object is intersected by a division plane (the circle in Figure 3.3) then the synchronization is achieved by first locking the parent areanode that contains the object (*A3* in Figure 3.3), then locking the object, and at the end unlocking the parent areanode so that other threads can get the objects they interact with.

In the parallel Quake server, execution of a game cycle is a repeated pattern of *input-compute-output* operations on memory locations that are read and written. Some of these memory locations are shared for concurrent writing or reading by multiple threads and therefore access to these objects is serialized by combining fine and coarse grain locking schemes. Coarse grain locks do not exploit the parallelism optimally whereas fine grain locks have complex implementations. Since the goal of TM is to make writing efficient parallel applications as easy as using a global lock, the parallel Quake server seems a good candidate workload to demonstrate the practical use of Transactional Memory in place of coarse grain locks to unlock parallelism and in place of fine grain lock to simplify implementation and reduce programming effort.

3.3.2 Using Transactions

The lock-based parallel implementation of the Quake server was transactified using a prototype version of the Intel C compiler with STM support [30; 124]. In AtomicQuake all the lock-based synchronization was replaced with `atomic` blocks. Although this process seems to be straightforward it ended to be rather complicated. Quake has many shared objects and also locks which protect them. First it was necessary to understand the association between the locks and the shared data structure (i.e. which locks protect which variable). Second, in some places (i.e. region based locking) to avoid deadlock special locking protocols were used and understanding these protocols was rather difficult. Third, in most cases the use of `lock` and `unlock` operations did not match the block structure of `atomic` blocks and it was necessary to make changes in the code without changing the game logic. All these required a careful code analysis and deep understanding of the code written in 27,400 lines of C code. All the effort cost 12 man months.

In Atomic Quake there are 63 atomic blocks in total. About 98.4% of the request processing phase (RP in Figure 3.2) executes in transactions and the request processing part of the program constitutes 63% of the total program execution when the server is fully loaded (see Figure 3.2). Almost all `atomic` blocks have function calls inside. The static call graph trees for the `atomic` blocks have about 4 levels and 20-25 nodes. There are nested transactions reaching

up to 9 levels depth and `atomic` blocks with recursive function calls. In some `atomic` blocks there were calls to the standard library, particularly the string manipulation routines that were later implemented manually. Inside some `atomic` blocks there were also calls to functions in the math library that were declared to be `tm_pure`[1] (see Section 2.1).

Until now this section introduced the programming abstractions provided by the compiler that was used. Now it continues with a discussion how transactions simplify the structure of accesses to the core areanode data structure (Section 3.3.2.1). Next are presented cases where a basic transactional model is less effective: non-block-structured critical sections (Section 3.3.2.2), condition synchronization (Section 3.3.2.4), IO operations within transactions (Section 3.3.2.5), error handling within transactions (Section 3.3.2.6), and cases where data changes between transactional and non-transactional accesses (Section 3.3.2.7).

3.3.2.1 Where Transactions Fit

Transactional Memory makes writing parallel code simple when there are many shared objects that would otherwise be guarded with a separate lock - a scheme that is referred as fine grain locking. In this case the programmer has to keep track of 1) the objects that might be attempted concurrent access; 2) the locks associated with these objects; and 3) the order of acquiring locks to avoid deadlock. However, when using transactions this becomes a trivial task, which requires the programmer to identify the part of the code that has to execute atomically and not bother with remembering the many shared variables and the locks guarding them. Uses of this kind cannot be seen in small applications as they have few objects but in applications like the parallel Quake server that have hundreds of locks, transactions save. Figure 3.4 shows an example code fragment where the object is first locked based on its type and then picked up. This code is trivial to implement with transactions and the transactional equivalent is shown in Figure 3.5.

[1] `tm_pure` is used to declare a function as not having side effects such as `sin`, `cos`, and can be called safely inside a transaction. In this case the compiler does not generate a transactional clone of the function.

```
 1: switch(object->type) { /* Lock phase */
 2:    KEY: lock(key_mutex); break;
 3:    LIFE: lock(life_mutex); break;
 4:    WEAPON: lock(weapon_mutex); break;
 5:    ARMOR: lock(armor_mutex); break
 6: };
 7:
 8: pick_up_object(object);
 9:
10: switch(object->type) { /* Unlock phase */
11:    KEY: unlock(key_mutex); break;
12:    LIFE: unlock(life_mutex); break;
13:    WEAPON: unlock(weapon_mutex); break;
14:    ARMOR: unlock(armor_mutex); break
15: };
```

Figure 3.4: Per-object locking.

Things turn to be more complicated when it is about to lock just a part of a collection data structure such as a tree. Then the programmer should implement a logic for locking and unlocking the required region by using supporting data structures such as stacks and queues. In Quake such kind of region locking is utilized quite a lot and consists of locking leaf nodes in areanode tree (see Figure 3.3). When a player moves or shoots, the places in the virtual world that the player can be or respectively the bullet can go though are first identified by a lightweight simulation and then the areanodes that map to these locations are locked and the operation carried out. To demonstrate the complexity of such type of fine grain locking Figure 3.6 gives a simplified example which locks the leaf nodes in a tree. The logic for acquiring locks uses supporting data structures such as stack. In addition to this logic in Quake a lightweight simulation is performed to identify the leafs to be locked. Implementing fine grain locking of this kind by using transactions is straightforward, where the critical section should just be put in atomic block as in Figure 3.5. Using transactions eliminates the necessity to perform lightweight simulation because the transactional logs implicitly keep record of accessed memory locations.

```
1: atomic
2: {
3:    pick_up_object(object);
4: }
```

Figure 3.5: Solution to the per-object locking with TM.

```
 1: /* Start locking leafs*/
 2: lock(tree.root);
 3: stack.push(tree.root);
 4: while (!stack.is_empty()) {
 5:    parent = stack.pop();
 6:    if (parent.has_children()) {
 7:       for (child = parent.first_child();
 8:            child != NULL; child.next_sibling()) {
 9:          lock(child);
10:          stack.push(child);
11:        }
12:        unlock(parent);
13:    }
14: } /* End locking */
15:
16: <move or shoot>
17:
18: /* Start unlocking */
19: if (tree.root.has_children()) {
20:    lock(tree.root);
21:    stach.push(tree.lock);
22: } else {
23:    unlock(tree.root);
24: }
25: while (!stack.is_empty()) {
26:    parent = stack.pop();
27:    for (child = parent.first_child();
28:         child != NULL; child.next_sibling()) {
29:       if (child.has_children()) {
30:          lock(child);
31:          stack.push(child);
32:       }
34:       else { // This is a leaf
35:          unlock(child);
36:      }
37:    }
38:    unlock(parent);
39: }
40: /* End unlocking */
```

Figure 3.6: Fine grain locking of areanode tree's leafs.

3.3.2.2 Non-Block-Structured Critical Sections

Because of the irregular parallelism, the transactification of the parallel Quake server was not as easy as just replacing a lock operation with "`atomic{`" and unlock with "}". First, it was necessary to dedicate significant effort to find the shared objects, understand their purpose and where in the code they are accessed concurrently. Most of the time it was necessary to change the code guarded by unstructured locks so that the critical section can be surrounded with an `atomic` block. Example source code with the described characteristics is shown in Figure 3.7. In the given example, the use of lock and unlock operations is unstructured, meaning that it does not match the structured block syntax of the `atomic` blocks. Transactifying this code fragment in a way that the `atomic` blocks match exactly the critical sections defined by the locks happens to be considerably difficult task and cannot be carried without understanding the logic in the code. The statements that make transactification difficult are <statements4> and <statements7>. Our solution to this particular example is shown in Figure 3.8. In the provided solution, two additional variables are introduced one per each `if` block that has unlock and moved <statements4> and <statements7> to the end of the `for` loop as their execution is guarded by the added flag variables. These statements are executed only if the corresponding if-flags are set to true. The provided solution also increases the complexity in the conditional logic. This solution is not necessary applicable for all the similar cases as the changes in the code depend on the code itself. The given example becomes much more complicated when there are more similar if blocks or new variables declared in <statements4> or <statement7>. In this case the declaration of these variables will have to be hoisted so that they are in scope. The given example shows a particular case where transactions with their current syntax does not fit well. If we had a way to explicitly commit the transaction, such as a `commit` keyword, then solving this problem would be as easy as just replacing `unlock` operations in lines 7 and 15 in Figure 3.7 with the `commit` keyword. Of course, if this application was written from scratch rather than ported then the developer most likely wouldn't pay much attention to the granularity and most likely would use a single `atomic` block starting at line 3 and ending at line 24 in Figure 3.7.

```
1: for (i=0; i<sv_tot_num_players/sv_nproc; i++){
2:    <statements1>
3:    LOCK(cl_msg_lock[c - svs.clients]);
4:    <statemnts2>
5:    if (!c->send_message) {
6:       <statements3>
7:       UNLOCK(cl_msg_lock[c - svs.clients]);
8:       <statements4>
9:       continue;
10:   }
11:   <stamemnts5>
12:   if (!sv.paused && !Netchan_CanPacket (&c->netchan)) {
13:      <statmenets6>
14:      UNLOCK(cl_msg_lock[c - svs.clients]);
15:      <statements7>
16:      continue;
17:   }
18:   <statements8>
19:   if (c->state == cs_spawned) {
20:      if (frame_threads_num > 1) LOCK(par_runcmd_lock);
21:      <statements9>
22:      if (frame_thread_num > 1) UNLOCK(par_runcmd_lock);
23:   }
24:   UNLOCK(cl_msg_lock[c - svs.clients]);
25:   <statements10>
26: }
```

Figure 3.7: Unstructured use of locks.

But almost in all cases we tried to adhere to the original multithreaded implementation, since based on the existing research, long transactions have performance impact on the overall execution [130] and coarsening the critical sections might have negative impact on the performance.

3.3.2.3 Thread Private Storage

Another problem was relevant with the use of thread private data that is set with `pthread_setspecific` and retrieved with `pthread_getspecific` APIs shown in Figure 3.9. Here, the user defined thread id is stored in a thread private data area. When a function called inside an atomic block makes a call to the `pthread` API causes the transaction to serialize (transaction becomes irrevocable [126]). For similar use cases, tool developers may consider implementing support for thread private data, such as declaring a thread private variable, which would reduce the

```
 1: bool first_if = false;
 2: bool second_if = false;
 3: for (i=0; i<sv_tot_num_players/sv_nproc; i++){
 4:    <statements1>
 5:    atomic {
 6:       <statemnts2>
 7:       if (!c->send_message) {
 8:          <statements3>
 9:          first_if = true;
10:          } else {
11:             <stamemnts5>
12:             if (!sv.paused && !Netchan_CanPacket(&c->netchan)){
13:                <statmenets6>
14:                second_if = true;
15:             } else {
16:                <statements8>
17:                if (c->state == cs_spawned) {
18:                   if (frame_threads_num > 1) {
19:                      atomic {
20:                         <statements9>
21:                       }
22:                    } else {
23:                       <statements9>;
24:                    }
25:                }
26:             }
27:          }
28:    }
29:    if (first_if) {
30:       <statements4>;
31:       first_if = false;
32:       continue;
33:    }
34:    if (second_if) {
35:       <statements7>;
36:       second_if = false;
37:       continue;
38:    }
39:    <statements10>
40: }
```

Figure 3.8: Unstructured use of locks - TM equivalent.

```
 1:  void foo1() {
 2:     atomic {
 3        foo2();
 4:     }
 5:  }
 6:
 7:  __attribute__((tm_callable))
 8:  void foo2() {
 9:     int thread_id = pthread_getspecific(THREAD_KEY);
10:     /* Continue based on the value of thread_id */
11:     return;
12:  }
```

Figure 3.9: Thread ID problem.

effort involved in porting lock based applications into transactions. It is useful to note that an alternative work around for this problem would be to declare function `pthread_getspecific` as `tm_pure` but initially with the first compiler we used there was no support with semantics of `tm_pure`.

3.3.2.4 Condition Synchronization

In the parallel Quake server, there is moderate use of conditional variables for synchronization. The STM compiler which was used did not implement appropriate primitives that can help replace the conditional variables and their associated locks by making reasonable changes in the code. To implement conditional synchronization in Transactional Memory, we need the retry language construct with the semantics defined by Harris et. al. [54]. Figure 3.10 shows how the `pthread` conditional synchronization would be implemented with the retry keyword.

3.3.2.5 IO and Irrevocability Inside Transactions

In almost every atomic block there is IO printing information messages on the screen (e.g. player connected, player died). It was necessary to comment some part of the code and in other to declare the IO functions as `tm_pure` otherwise many transactions were executing in irrevocable mode [75; 116; 126]. In our case hoisting the IO out of the `atomic` block might be considered impossible since it is done in functions called within the `atomic` block. For this particular pattern

```
------------- locks -------------
1: pthread_mutex_lock(mutex);
2: <statements1>
3: if (!condition)
4:    pthread_cond_wait(cond, mutex);
5: <statements2>
6: pthreda_mutex_unlock(mutex);
.
 ------------ retry --------------
1: atomic {
2:    <statements1>
3:    if (!condition)
4:       retry;
5:    <statements2>
6: }
```

Figure 3.10: Implementing conditional synchronization with retry.

it would be useful if there was a way, for example a key word like `escape_tm`, that would let us declare inside the `atomic` block statements to be executed non transactionally. Similar results would be achieved if we put the particular non-transactional code in a function declared as `tm_pure` and call this function inside the atomic block in place of the statements. But the solution of having a keyword like `escape_tm` would be more constructive and serve its purpose.

3.3.2.6 Error Handling Inside Transactions

When transactifying the source code, there were many places where it was necessary to handle errors inside transactions. In some cases it was considerably easy to do so but in others the existing language extensions were not expressive enough. Every time we tried to adhere to the approach chosen by the compiler developers for C++ exceptions which when error occurs first try to commit the transaction and then handle the error. For example, Figure 3.11 shows a function that has to handle a critical system error. The solution in AtomicQuakeis is given in Figure 3.12. The calls to function `Sys_Error` are taken outside the `atomic` block. Inside the transaction the type of the value is saved in a local variable `error_no`. The transaction is forced committed when the execution reaches the `break` statement. After the transaction commits, the error is examined and the proper action is taken.

```
 1 void Z_CheckHeap (void)
 2 {
 3   memblock_t *block;
 4   LOCK;
 5   for (block=mainzone->blocklist.next;;block=block->next){
 6   if (block->next == &mainzone->blocklist)
 7   break; // all blocks have been hit
 8   if ( (byte *)block + block->size != (byte *)block->next)
 9     Sys_Error("Block size does not touch the next block");
10   if ( block->next->prev != block)
11     Sys_Error("Next block doesn't have proper back link");
12   if (!block->tag && !block->next->tag)
13     Sys_Error("Two consecutive free blocks");
14   }
15   UNLOCK;
16 }
```

Figure 3.11: Error handling - lock based code.

The given example becomes much more complicated when function `Z_CheckHeap` is called inside another transaction. In this case, it would be necessary to call function `Sys_Error` outside the outermost transaction. To be able to do this are required mechanism such as commit handlers. Using commit handlers one would be able to dynamically tell the compensating actions that should be taken. For now we ignored the complicated cases like this by just letting the runtime switch to irrevocable mode.

It is debatable whether committing a transaction on error is right choice. What would happen if the transaction aborts instead of commits? There outcomes might be possible: 1) the error was repaired; 2) we lost detecting a hidden bug and; 3) worse getting the system into inconsistent state. In managed languages such as C# and Java, this kind of approach for handling errors might compromise the memory consistency and open security wholes. This simple example concludes that error handling in transactional code require deeper analysis and primitives helping to detect and recover from errors.

In the Quake code there are patterns of error handling that can gently benefit from the failure atomicity. An example code fragment part of the implementation of the request dispatcher function is given in Figure 3.13. In lines 18 and 25 is called function `PR_RunError` that prints the stack trace and terminates the

```
 1 void Z_CheckHeap (void) {
 2 memblock_t *block;
 3 int error_no = 0;
 4 atomic{
 5    for (block=mainzone->blocklist.next;;block=block->next){
 6       if (block->next == &mainzone->blocklist)
 7          break; // all blocks have been hit
 8       if ((byte *)block + block->size !=
 9          (byte *)block->next; {
10          error_no = 1;
11          break; /* makes the transactions commit */
12       }
13       if (block->next->prev != block) {
14          error_no = 2;
15          break;
16       }
17       if (!block->tag && !block->next->tag) {
18          error_no = 3;
19          break;
20       }
21    }
22 }
23 if (error_no == 1)
24   Sys_Error ("Block size does not touch the next block");
25 if (error_no == 2)
26   Sys_Error ("Next block doesn't have proper back link");
27 if (error_no == 3)
28   Sys_Error ("Two consecutive free blocks");
29 }
```

Figure 3.12: Error handling - in a transaction.

```
 1: void PR_ExecuteProgram (func_t fnum, int tId){
 2:    f = &pr_functions_array[tId][fnum];
 4:    pr_trace_array[tId] = false;
 5:    exitdepth = pr_depth_array[tId];
 6:    s = PR_EnterFunction (f, tId);
 7:    while (1){
 8:       s++;    // next statement
 9:       st = &pr_statements_array[tId][s];
10:       a = (eval_t *)&pr_globals_array[tId][st->a];
11:       b = (eval_t *)&pr_globals_array[tId][st->b];
12:       c = (eval_t *)&pr_globals_array[tId][st->c];
13:       st = &pr_statements[s];
14:       a = (eval_t *)&pr_globals[st->a];
15:       b = (eval_t *)&pr_globals[st->b];
16:       c = (eval_t *)&pr_globals[st->c];
17:       if (--runaway == 0)
18:           PR_RunError ("runaway loop error");
19:       pr_xfunction_array[tId]->profile++;
20:       pr_xstatement_array[tId] = s;
21:       if (pr_trace_array[tId])
22:           PR_PrintStatement (st);
23:   }
24:   if (ed==(edict_t*)sv.edicts && sv.state==ss_active)
25:       PR_RunError("assignment to world entity");
26:   }
27: }
```

Figure 3.13: Using failure atomicity to recover from critical error.

process. In this particular case the code from lines 2 to 26 including can be wrapped in an atomic block and instead of calling `PR_RunError`, abort the transaction. The abort would restore the original values of the global state stored in `pr_global_array` and `pr_global` and the execution will continue from line 27 which is the end of function `PR_ExecuteProgram`. The effect of this usage will be that client's request will not be processed as if it is lost on the network and the server will continue running. There are many similar uses like this but failure atomicity cannot be applied to all of them because it is important that the execution of the program can proceed safely.

3.3.2.7 Privatization

When transactifying the Quake source code we encountered several instances of memory privatization [56]. An example case is shown in Figure 3.14. In the

```
1: void* buffer;
2: atomic {
3:    buffer = Z_TagMalloc(size, 1);
4: }
5: if (!buffer)
6:     Sys_Error("Runtime Error: Not enough memory.");
7: else
8:    memset(buf, 0, size);
```

Figure 3.14: Example privatization.

given example, in the atomic block, the function `Z_TagMalloc` allocates a memory block to variable named buffer. The allocated memory block is not supposed to be returned by a subsequent call to `Z_TagMalloc` until it is not freed and therefore is safely modified outside the atomic block.

3.3.2.8 Call Graph Structure in Atomic Blocks

To give an insight to the reader about the complex call graph structure within atomic blocks we Figure 3.3.2.8 shows the call sequence in atomic block form `SV_RunCmd function`. The nodes drawn with clouds are calls to other functions that has as complex call graph structure as the current one. In this call graph the back edges from recursive calls are not shown.

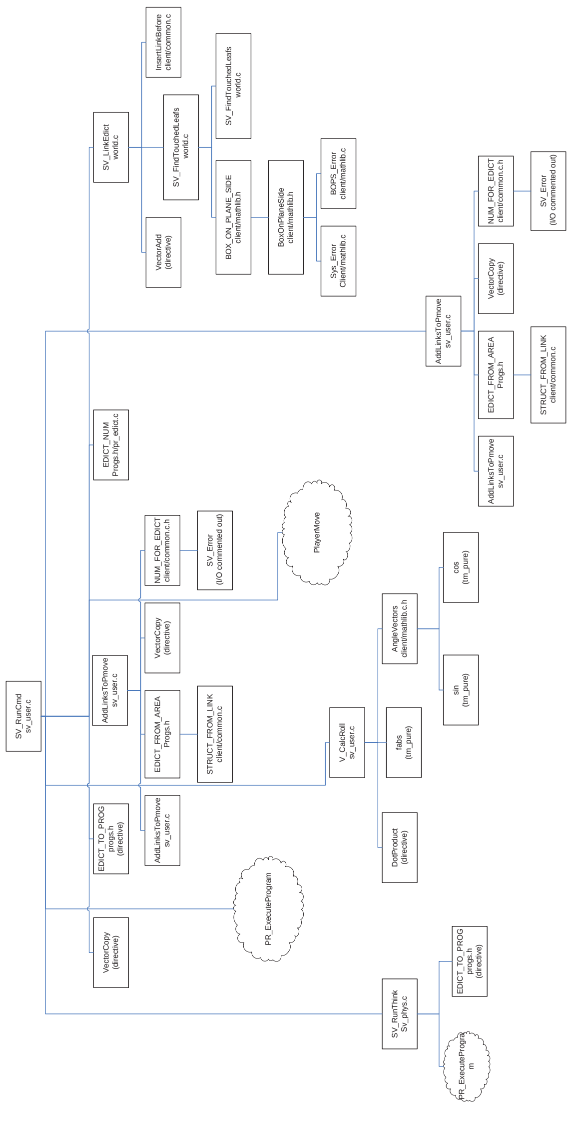


Figure 3.15: An example call graph in the `atomic` block from function SV_RunCmd.

3.3.3 Experimental Results

This section discusses the performance of AtomicQuake. Section 3.3.3.1 describes the experimental methodology. Section 3.3.3.2 presents the global application characteristics. Section 3.3.3.3 presents per-`atomic` block characteristics.

3.3.3.1 Experimental Methodology

To carry out the experiments for this research work we slightly modified the synchronization logic of the original version which enforces all the server threads to start processing the received client requests simultaneously. The modification that we added is shown on Figure 3.16. This scenario is equivalent to a fully loaded server that continuously receives requests and has to process them. Otherwise, without this change, to load the server we would have to connect about 460 clients and a game session of such scale would require a lot of configuration and computer resources which are difficult to find in a research laboratory. With the modified version of the Quake server we setup a game sessions with 1, 2, 4 and 8 threads and connected one client to each server thread. Each client was running on a different commodity PC complying to the Quake client's system requirements. Because of the repeatability of the experiments, clients were configured to run a prerecorded trace of player actions and no human player was playing. The statistical data was collected after all the clients get connected and continued until the server receives and process 1000 requests from each client[1]. Each experiment was executed 4 times and results averaged. Every experiment used the same virtual map. Because of runtime problems we could only use the smallest map which consists of only one room suitable for 1-2 players. We couldn't use other maps because the STM version of the code was crashing at the initialization.

The experiments were carried on Dell PE6850 workstation with 4 dual core x64 Intel Xeon processors with 32KB IL1 and 32KB DL1 private per core, 4MB L2 shared between the two cores on die, 8MB L3 shared between all cores, and 32GB RAM. The installed operating system was Suse 11.0. The prototype version

[1]Except for the results that are generated by the STM runtime itself that we don't have control over.

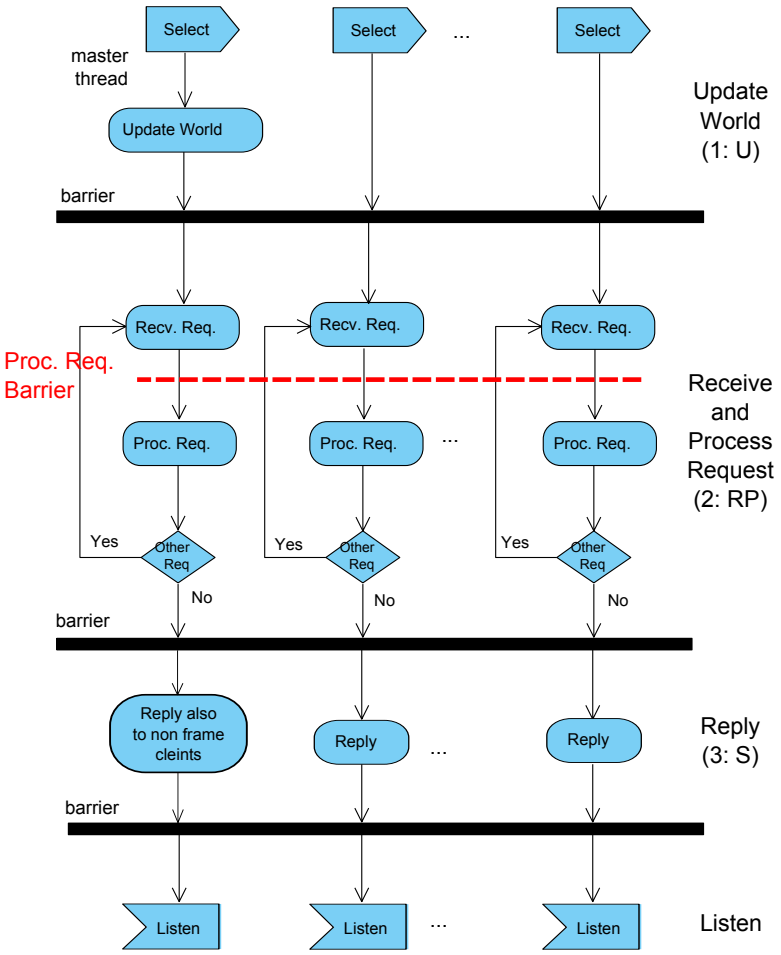


Figure 3.16: An additional barrier to enforce all the threads start processing client requests simultaneously.

of Intel C compiler with STM support [30; 124] was used to compile the code with compiler optimizations disabled (-O0), because the compiler was failing to compile the source code with optimizations turned on.

3.3.3.2 Application Characteristics

Figure 3.17 shows the speedup of AtomicQuake normalized to the single threaded execution and Figure 3.18 show how AtomicQuake scales (every execution is normalized to itself). In this experiment LOCK is the lock-based version of Quake, STM is the STM version and STM_LOCK is the STM version of Quake where every `atomic` block is protected by a global reentrant lock (i.e. the lock is acquired before entering the `atomic` block the released after exiting the `atomic` block. The rationale in doing this, is to have a realistic base line of a global lock where the instrumentation overhead of transactions is counted in. Because, unlike the recent research results obtained with μbenchmarks and kernel application that report STM overhead less than x2.5, in our experiments the STM overhead on single threaded runs was x4-x5, resulting in meaningless results when comparing STM with pure-lock based implementations. In Figure 3.17 shows the speedup computed from the throughput. Because all the threads start to process the client requests at the same time, we can assume that measured results will match those when the server is 100% loaded - the requests' queue is not empty.

In Figure 3.18, all versions scale up to 4 threads and at 8 threads the STM version is saturated. With 8 threads the server is saturated and the performance of the STM version is worse than the single threaded base line due to the many aborts. The results also show that the scalability of the STM version version is limited by the aborts, whereas the scalability of STM_LOCK by the serialized execution of the critical section. The map that we used in the game session is small and represents high-conflict scenarios where the players interact with each other all the time. Unfortunately, because of issues in the tool set that we used, we couldn't run experiments with larger maps. Also, it is noteworthy to say that the transactional Quake server may perform better if not fully loaded which may result in less conflicts because of non-interleaved execution of the critical sections.

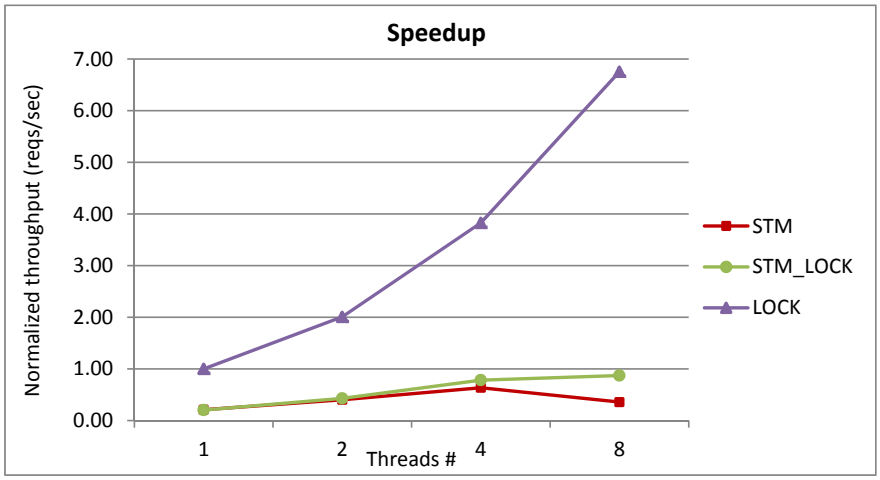


Figure 3.17: AtomicQuake speedup. Results are normalized to the single threaded execution.

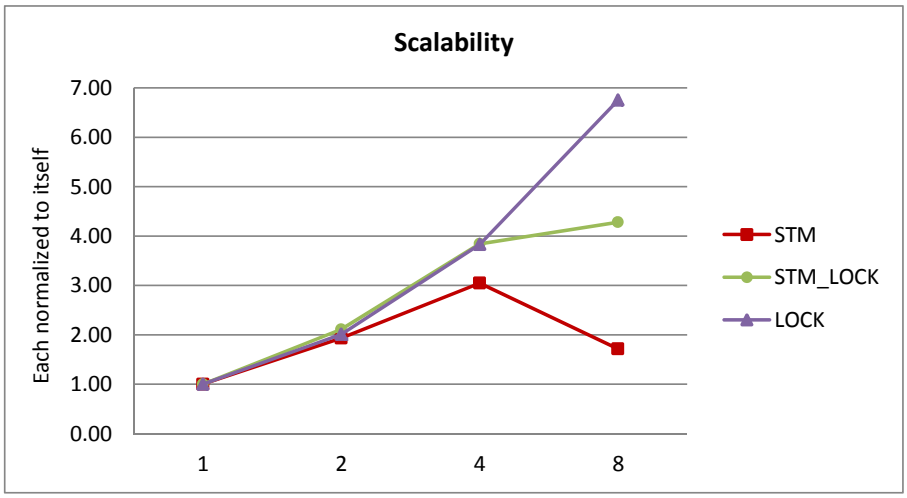


Figure 3.18: AtomicQuake scalability. Every plotted result is normalized to itself. This figures shows better how AtomicQuake scales.

Threads	Transactions	Aborts		Serialized Tx
		Num	%	
1	36 667	0	0.00%	17
2	75 824	241	0.42%	31
4	166 000	2 612	1.58 %	85
8	477 519	76 771	25.50%	237

Table 3.1: Transactional characteristics.

Table 3.1 summarizes the transactional characteristics of Quake. With 2 threads there are almost no aborts and the rate of aborts increases exponentially with the number of threads. The high rate of aborts and consequently wasted computation time is the reason why the Atomic Quake server (STM) scales poorly. Since conflicts has such a big impact on the performance, this is also a signal that more research should be done on conflict avoidance, detection and resolution in STM libraries.

Table 3.2 shows a summary about the read and write set for all the transactions in the application. The reported results for the read and write sets indicate the number of bytes read or written from the beginning to the end of the transaction including those accumulated during transaction re-executions on abort. In AtomicQuake reads dominate over writes and the average size of the read and write set is small enough to fit in the caches of modern CPUs. This also makes the AtomicQuake a good workload for evaluating the performance of HTMs. However, there are also transactions with large read set up to 1.5MB and write set up to 344KB that would overflow the caches and would require mechanisms for virtualization.

3.3.3.3 Per-Atomic Block Characteristics

Table 3.3 shows per `atomic` block statistics obtained from a single-threaded execution of AtomicQuake. Entries for the `atomic` blocks that were not executed in our test workload (e.g. walking in water) are omitted.

These results suggest that a small number of `atomic` blocks make up the bulk of the transactional workload (i.e. `atomic` blocks 56, 57, 58, 60, 61). These `atomic` blocks are located in functions in the critical path of the RP phase. The

Threads	Read Set (bytes) Accumulated					Write Set (bytes) Accumulated				
	Min	Avg	Max	Total	Reads	Min	Avg	Max	Total	Writes
1	8	490	53 566	18 214 226	83%	0	98	11 161	3 639 952	17%
2	8	540	172 508	40 907 196	83%	0	115	47 784	8 737 623	18%
4	4	575	181 740	95 505 459	81%	0	131	52 032	21 737 915	19%
8	4	798	1 591 946	381 290 019	81%	0	183	352 640	87 837 969	19%

Table 3.2: [AtomicQuake – read and write set sizes.] The reported results for the read and write sets indicate the number of bytes read or written from the beginning to the end of the transaction including those accumulated during transaction re-executions on abort.

second group of transactions execute much less frequently; e.g. they include an example where the server sends a message to a client that has expended all their weapons.

Also, the results in Table 3.3 suggest both that `atomic` blocks have different transactional characteristics and that the different executions of the same `atomic` are different. For example, there are short transaction consisting of just few hundreds of CPU cycles and there long running transactions which span up to 1 milliseconds. In another example, the execution of the same `atomic` block can read or update different set of variables. It is worth noting that the most frequently executed `atomic` block is a simple read-only non-nesting example which seems amenable to hardware implementation in a hybrid implementation.

ID	Tx#	Nest	Dynamic Length (CPU Cycles)				Read Set (bytes)				Write Set (bytes)			
			Total	Min	Max	Avg	Total	Min	Max	Avg	Total	Min	Max	Avg
56	26 962	0	172 872 572	288	112 832	6 412	1 328 536	20	104	49	0	0	0	0
60	5 931	0	5 810 152	224	41 552	980	76 212	12	640	13	928	0	116	0.16
61	1 095	0	20 573 540	4 560	49 984	19 208	723 474	88	776	661	90	84	84	84
59	1 042	0	3 117 844	1 520	39 344	2 999	29 176	5	28	28	16 672	16	16	16
57	1 038	5	401 502 152	288 704	522 528	387 552	10 963 719	7 614	15 490	10 562	2 592 367	1 680	3 656	2 497
58	1 002	1	134 949 344	87 056	1 341 504	134 949	5 054 282	3 028	53 566	5 044	931 445	548	11 161	930
15	3	0	67 660	720	48 176	1 735	96	32	32	32	18	6	6	6
5	2	0	99 988	592	36 384	1 923	64	32	32	32	10	5	5	5
22	2	1	43 632	12 176	35 504	21 816	72	36	36	36	128	64	64	64
36	2	0	40 476	6 800	44 880	20 238	249	108	141	125	55	22	33	28
38	2	0	71 368	2 144	31 504	4 461	90	44	46	45	26	12	14	13

Table 3.3: Statistics for each `atomic` block from a single-threaded execution. Transactions that are not executed are not shown.

3.4 WormBench

WormBench is a parameterized synthetic workload for TM. Its design accounts for the common synchronization problems that exist in multi-threaded applications. WormBench is easily configurable and can be set to have desired runtime characteristics. This feature is found useful in reproducing pathological execution which otherwise happen once in a while in the real workloads. Also, the same flexibility can be used to model the runtime characteristics of a real application.

WormBench is implemented in C# and does not depend on a particular STM or HTM interface because the critical sections in the code are expressed in terms of the language-level `atomic` blocks. It assumes that the compiler or runtime system translates these into the appropriate concurrency control operations on a TM implementation. This way WormBench can be used also to test the effectiveness of optimizations performed by TM-enabled compilers [10; 39; 55; 89; 124].

The idea of WormBench is inspired by the popular Snake game (see Figure 3.19). In the application several worms, each driven by a dedicated thread, move within a shared environment - BenchWorld (abstraction of a matrix). Every move consists of several critical operations accompanied by a computation. Worms can be grouped so that they recreate complex synchronization scenarios. By changing the parameters of the applications such as the type of performed computation, the size of the BenchWorld and the Worm, one can make a different run configuration which has the desired transactional and runtime characteristics.

The motivation of building WormBench is to help TM researchers easily create transactional workloads that they can use to verify and evaluate the efficiency of their TM systems and the compiler infrastructure that sits between the programming language and the TM. Using WormBench, one can develop a set of representative run configurations that has the transactional behavior of a typical multi-threaded application. Then use these run configurations as a baseline to compare different TM systems among each other and against the lock based version. Also, as being general enough, WormBench can be configured as a workload to stress a low level implementation detail in the TM system such as frequent read set overflows. During this dissertation WormBench was many times used to

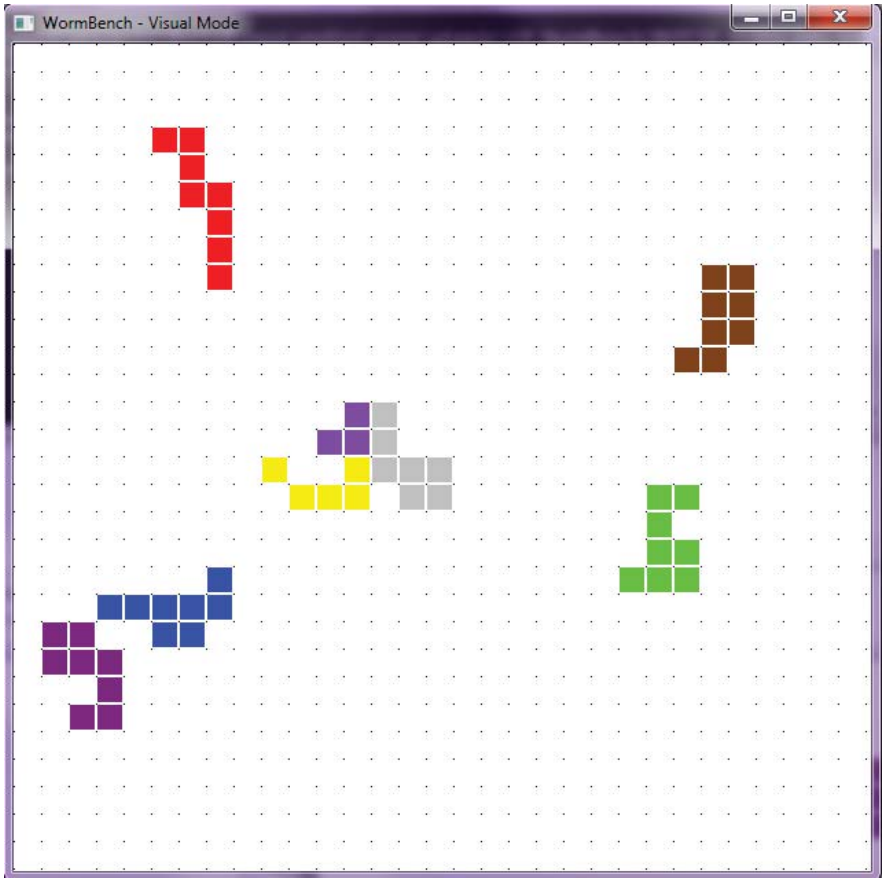


Figure 3.19: Screenshot from WormBench

stress the certain components and extensions of the Bartok-STM (i.e. profiling framework extension, logging indirect variables etc.)

3.4.1 Requirements for a Synthetic TM Workload

Because TM is about concurrency control, the main requirements for a representative synthetic TM workload should include the common synchronization problems that exist in multi-threaded applications. In this way, we would be able to see how a given synchronization problem is solved by conceptually different techniques – locks which are blocking versus transactions which are non-blocking and compare them against each other. And also, to be able to compare different TM systems, it is required that a representative workload should consider the essential features of the TM system. This section discusses the synchronization problems and the TM relevant metrics that should be considered when building a representative workload or a suit of workloads to evaluate Transactional Memory systems.

3.4.1.1 Synchronization Problems

The necessity of having concurrency control is because of the common synchronization problems that exist in multi threaded applications. The typical synchronization problems that can be seen in these applications and that a representative synthetic TM workload should have an instance of, are:

- Object access serializability [13] - managing a concurrent access to a shared data. This is the typical scenario when we guard the access of a shared variable by lock;
- Barrier synchronization - making group of threads to wait at certain point of execution until all (or group) of them arrive there;
- Two phase locking and its derivatives [117] - a locking protocol which attempts to provide the efficiency of fine grain locking and avoiding deadlock by enforcing a given pattern;

- Dining philosophers [118] - is a synchronization problem that demonstrates the deadlock problem;
- Multiple granularity locking [70; 135] - a fine grain locking technique used to lock a region in a hierarchical data structures like trees;

3.4.1.2 Metrics

To be able to compare different TM system against each other and also TM systems against lock-based implementations, a representative workload application should clearly identify a set of metrics that can be used to quantitatively evaluate the performance of different TM systems. These metrics should source from the application and not be specific to a particular design or implementation style of any TM system (HTM or STM). Based on the metrics used in the existing TM research, we decided to collect the following runtime metrics in an application:

- Execution time of the application;
- Number of entered critical sections (i.e. atomic blocks);
- The ratio between reads and writes (e.g. 90% reads and 10% writes);
- Size of the accessed data structures;
- The execution time spent while in a critical section (short transactions vs. long transactions);
- Number of successfully committed transactions;
- Number of reads and writes per transaction;
- Prevalent type of operations in the application (IO, CPU, memory); and
- Locality of memory references (spatial vs. temporal).

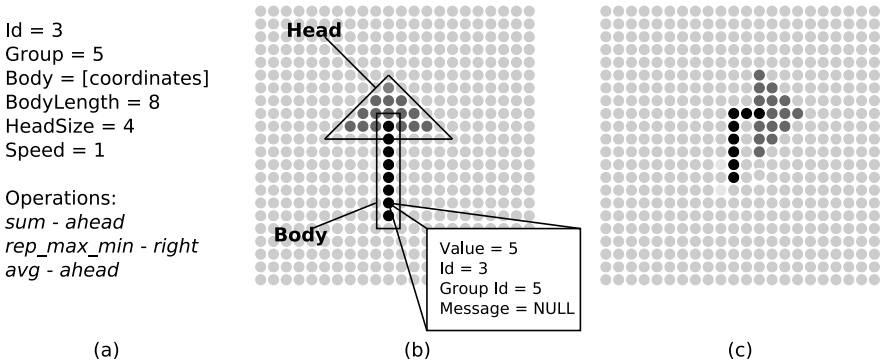


Figure 3.20: The main components in the WormBench application. (a) run configuration; (b) the position of the worm before performing the operations in the run configuration; (c) position of the worm after performing the operations in (a).

3.4.2 Design and Implementation

The idea for WormBench is inspired from the Snake game (see Figure 3.4.1). The application has two main data structures - *BenchWorld* and *Worm*. In the application, several Worms move in the BenchWorld and execute worm operations from an user specified stream (see Figure 3.20). Each cell in BenchWorld is a *BenchWorldNode* struct which packs several data: (1) a value of the node, (2) the reference to the worm that is on this cell, (3) a reference to the group to which the worm on this cell belongs to, (4) and a message for the next worm that will pass from this cell.

Worms are active objects meaning that every Worm object is associated with one thread. A Worm object has several attributes: id, group, speed, body, and head. *Id* is a unique identifier to distinguish the worm from the other worms, *group* is a reference to a Group object that groups several worm objects together. The rationale behind the notion of group is to be able to create synchronization scenarios where several worms act together to achieve a common task. The *speed* attribute is used to tell how fast the worm to advance (e.g. 1 cell per move). The *body* of the worm is the set of the cells from the BenchWorld where the worm steps on. The *head* of the worm represents a set of nodes from the BenchWorld that the worm uses as input to every worm operation, and the result of the performed

computation is stored in a private buffer for verification purposes. Worms are initialized with a stream of *worm operations* (see Figure 3.20-a) that they should perform on every move. Every move is completed in three steps: (1) read the cells below the head, (2) perform a worm operation on the head values, (3) and move its body forward. Each of these three steps involves a critical operation and is either synchronized with an `atomic` block (TM system) or with a global lock (preset at compile time). Reading the values below the head of the worm involves computing the worm orientation and the head coordinates. When the head values are read, the next worm operation from the operations stream is applied to these head values and the produced result is stored in a private buffer for verification purposes. When it's time to advance forward, the worm updates the group field of every node constituting its body. In the transactional version of the benchmark this is a conditional `atomic` block which ensures that worms belonging to other groups cannot cross through each other. Every attempt of crossing would result in aborting the attacker transaction and blocking until the other worm moves its body out of the occupied node. The currently implemented worm operations that worms apply in step (2) are:

1. *sum* - sum all the values under the head; this operation is a basic computation and does not update the BenchWorld;

2. *average* - computes the average value of the cells under the worm's head; this operation is a basic computation and does not update the BenchWorld;

3. *median* - computes the median value among the cells under the worm's head; this is a search operation where values are first sorted and then median is found;

4. *min* - finds the minimum value among the cells under the worm's head; a basic search operation;

5. *max* - finds the maximum value among the cells under the worm's head; basic search operation;

6. *replace max with avg* - finds the maximum value and updates it with the average value; this operations is a combination between max and average operations. It involves a basic search and computation (only one node is updated);

7. *replace min with avg* - finds the minimum value and updates it with the average value; this operations is a combination between min and average operations. It involves a basic search and computation;

8. *replace median with avg* - finds the median value and updates it with the average value; this operations is a combination between median and average operations. It involves a little bit more complex search and computation;

9. *replace median with min* - finds the median and the minimum value from the worm's head and replaces; this operation is a combination between median and min operations and also involves a small updated to BenchWorld (two nodes are updated);

10. *replace median with max* - finds the median and the maximum value from the worm's head and replaces; this operation is a combination between median and max operations and also involves a small updated to BenchWorld;

11. *replace max with min* - replace the maximum and minimum in the head and write the changes to the BenchWorld so that they are globally visible;

12. *sort* - sort the values under the head and write the result to the BenchWorld; this operations involves a significant atomic update on the BenchWorld;

13. *transpose* - sort the values under the head and write the result to the BenchWorld; this operations involves a significant atomic update on the BenchWorld

14. *checkpoint* - the worm persists its current location (coordinates) within the BenchWorld to a file for later undo; this operation involves basic IO;

15. *undo* - this operation returns the body of the worm to the last checkpoint if any; this operation involves a basic IO;

16. *leave message* - leaves a message on a node to be read by the other worms; the rationale behind introducing this operations is allowing complicated interactions between the worms and this way instantiating the different synchronization problems described in Requirements section. The currently supported message is "goto node" that can be valid for a specific worm, for a group of worms or for all worms. When the "goto node" message is read by the intended recipient worm, it heads to the destination by following the shortest path and continuously applying worm operations on every move.

At the end of the execution, WormBench performs automatic correctness test (i.e. sanity check for the TM system). The correctness test is necessary to verify whether the TM system worked properly and consists of comparing the sum of the matrix at the end of the execution with the sum of the matrix that was at the initialization. When computing the sum of the matrix at the end of execution WormBench accounts for the modifications done by the *replace with average* operations. These modifications are stored in worms' private buffers.

WormBench is implemented in C# language by applying the concepts of object oriented programming and has a compact code base which consists of 940 lines. The code is implemented with two types of synchronization - *transactions* (`atomic` blocks) and *global lock*. The synchronization type can be selected at compile time. The average sizes of the shared objects is 70 bytes and have several fields which makes it favorable for TM systems that perform the versioning in object granularity (mostly STM), cache line and word granularity (mostly HTM). The primary performance evaluation metric in the BenchWorld application is the *throughput - the total number of moves per unit time*.

In behavior and synchronization, WormBench resembles the typical multithreaded applications where independent threads perform memory reads, do computation and update a given global state. An example could be a web server with dynamic content rendering. Where the requests of the clients are served by different threads as the memory is searched for cached pages and updated on the fly depending on the provided input by the client.

3.4.3 Runtime Characteristics

The transactional characteristics of the WormBench operations are given in Table 3.4 and Table 3.5. Both, Table 3.4 and Table 3.5, show respectively how the change on the Worm's body length and the head size affect the transactions' read (R) and write (W) set per each Worm operation. When the head size is constant and only the body length changes, the read set remains constant and the write set increases linearly [1]. On the other hand, when the body length is fixed to 1 and the head size changes, both the read and write sets are affected and the read set increases super-linearly [2]. Any combination of these operations with the body length and head size of the worms could give theoretically infinite number of TM specific runtime configurations.

Table 3.6 summarizes the execution distribution of the Worm operations for 4 different body length and head size setups ran over 800,000 moves. The first column is the worm operation, the second column is the execution distribution when the body length and head sizes are 1-1 (B[1.1] means body length is 1, H[1.1] means the head size is 1), the third column is for worms with body length and head size of 4-4, the fourth column is when the body length and head size is 8-8 and the fifth is when the body length and head size is randomly selected in range [1, 8]. Also, the increase in the head size is reverse-proportional to the WormBench throughput (execution time). Meaning that, by increasing the head size we can obtain longer transactions suitable to test STMs and by decreasing the head size we can obtain shorter transactions suitable to test HTMs. The relationship between the head size and the throughput can be seen in Figure 3.22 and Figure 3.23 discussed in more details in Section 3.4.4.

When WormBench starts, it is initialized with a *run configuration* provided as input by the user. The *run configuration* defines: (1) the size of the BenchWorld (the size of the underlying matrix) and its initialization, (2) a common stream of worm operations; (3) the number of worms to create; (4) and for each worm: id, group id, body size and the location of the body on the BenchWorld, head

[1] The exact rate of increase depends on the underlying TM system. In our case with every step the number of reads increases by 13.

[2] The super-linear increase in the read set is because the number of nodes below the head is n2 proportional.

Operation	1		2		4		8	
	R	W	R	W	R	W	R	W
sum	11	3	11	4	11	6	11	10
average	11	3	11	4	11	6	11	10
median	11	3	11	4	11	6	11	10
min	11	3	11	4	11	6	11	10
max	11	3	11	4	11	6	11	10
repl(max,min)	14	5	15	5	14	7	14	11
repl(min,avg)	14	5	15	5	14	7	14	11
repl(med,avg)	14	5	15	5	14	7	14	11
repl(med,min)	14	5	15	5	14	7	14	11
repl(med,max)	14	5	15	5	14	7	14	11
repl(max,min)	14	5	15	5	14	7	14	11
sort	16	4	16	5	16	7	16	11
transpose	16	4	16	5	16	7	16	11
checkpoint	11	3	11	4	11	6	11	11
undo	11	3	11	4	11	6	11	11

Table 3.4: The effect of the HeadSize on read and write.

Operation	1		2		4		8	
	R	W	R	W	R	W	R	W
sum	11	3	14	3	26	3	74	13
average	11	3	14	3	26	3	74	13
median	11	3	14	3	26	3	74	13
min	11	3	14	3	26	3	74	13
max	11	3	14	3	26	3	74	13
repl(max,avg)	14	4	17	4	29	4	77	4
repl(min,avg)	14	4	17	4	29	4	77	4
repl(med,avg)	14	4	17	4	29	4	77	4
repl(med,min)	14	4	17	5	29	5	77	5
repl(med,max)	14	4	17	5	29	5	77	5
repl(max,min)	14	4	17	5	29	5	77	5
sort	16	4	19	7	31	19	79	67
transpose	16	4	19	7	31	19	79	67
checkpoint	11	3	14	3	26	3	74	13
undo	11	3	14	3	26	3	74	13

Table 3.5: The effect of the BodyLength on read and write.

Operation	B[1.1]H[1.1]	B[4.4]H[4.4]	B[8.8]H[8.8]	B[1.8]H[1.8]
sum	0.42	0.43	0.19	0.31
average	0.42	0.27	0.32	0.43
median	0.84	3.65	9.35	5.14
min	0.32	0.59	0.28	0.37
max	0.42	0.59	0.33	0.54
repl(max,min)	1.36	0.71	0.43	0.74
repl(min,avg)	0.74	0.74	0.53	0.70
repl(med,avg)	2.52	4.79	11.41	6.69
repl(med,min)	2.10	0.59	0.63	0.93
repl(med,max)	2.73	5.01	11.19	7.10
repl(max,min)	2.52	5.26	11.39	7.12
sort	1.68	6.59	11.26	7.18
transpose	1.15	3.25	2.37	3.37
checkpoint	1.12	1.45	1.98	1.52
undo	1.06	1.32	1.85	1.49
TOTAL	19.39	35.24	63.52	42.60

Table 3.6: Execution time distribution of Worm operations.

size, speed, and a range from a common stream of worm operations that the worm has to perform on every move. By utilizing the summarized information in Table 3.4, Table 3.5 and Table 3.6, it is possible to directly control the read set, write set and the conflict rate. Also, assigning each worm a specific stream of operations to perform, it is possible to coarsely control the conflict rate between the transactions. For example, a stream of operations that leads all worms in a common point within the BenchWorld would result into a large number of aborts. Furthermore, by properly using the messaging and the groups, it is possible to recreate instances of the synchronization problems as described in Section 3.4.1.

3.4.4 Experimental Analysis

The overall behavior of the WormBench application depends on the run configuration passed as input by the user. The runtime characteristics of the application can be altered by tuning any of the following parameters:

- Size of the BenchWorld;
- Number of worms (number of threads);
- Body length of each worm;
- Head size of each worm;
- The number and type of worm operations that each worm has to perform while moving; and
- Synchronization type - atomic, lock.

By altering any of these configuration parameters it is possible to prepare a run configuration which runtime characteristics represent a typical multi-threaded application. In the same way WormBench can be configured to stress a particular aspect of the TM system such as many aborting transactions.

This section examines several run configurations. The purpose of this experiment is to study the relationship between the configuration parameters and the behavior of WormBench. Obtained results also include comparison between transactional memory and lock-based synchronization.

Experiments were carried on Dell PE6850 workstation with 4 dual-core x64 Intel Xeon processors with 32KB IL1 and 32KB DL1 private per-core, 4MB L2 shared between the two cores on-die, 8MB L3 shared between all cores, and 32GB RAM. During our experiments hyper-threading was enabled, thus having 16 logical CPUs. The operating system is Windows Server 2003 SP2. The processor scheduling and the memory management policies were adjusted to favor foreground applications instead of background services. To compile the WormBench source code we used Bartok compiler [55].

3.4.4.1 Description of the Run Configurations

In the WormBench experiments a single stream of 800.000 move operations was used. Both the operation type and the direction to move to were randomly generated with uniform distribution of the described worm operations (without leave message operation) and the three directions (ahead, left, right). To analyze the impact of the BenchWorld size 4 different BenchWorlds with 128x128, 256x256, 512x512, and 1024x1024 sizes were used. To analyze how the worm's body length and head size affect the execution four different (body length, head size) configurations were used - all the worms have body length and head size 1 (indicated as B[1.1]H[1.1]), all the worms have body length and head size 4 (B[4.4]H[4.4]), all the worms have body length 8 and head size 8 (B[8.8]H[8.8]), and all the worms have both body length and head size randomly generated in range [1, 8] (B[1.8]H[1.8]). To see how the worms initialization affect the execution, worms in large BenchWorld were initialized for a small BenchWorld. For example, worms in BenchWorld with size 1024x1024 were initialized for BenchWorld with sizes 128x128. As shown on Figure 3.21, worms initialized for smaller BenchWorld are relatively closer to each other and likely to be source of frequent conflicts. Combinations of all configurations were executed with 1, 2, 4, 8 and 16 worms (i.e. threads). This resulted in total of 80 combinations with 400 independent runs.

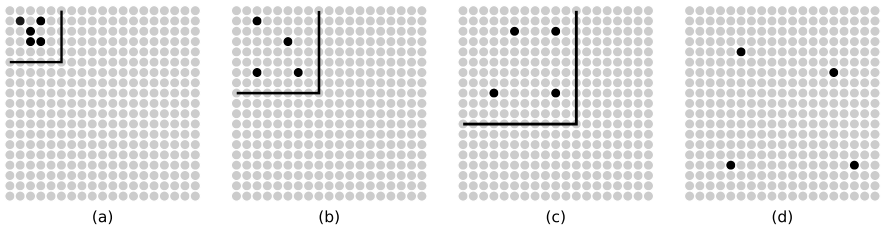


Figure 3.21: Using worms initialized for small BenchWorld in a large BenchWorld. (a) using worms initialized for 128x128 in 1024x2024; (b) using worms initialized for 256x256 in 1024x2024; (c) using worms initialized for 512x512 in 1024x2024; (d) using worms initialized for 1024x2024 in 1024x2024.

3.4.4.2 Results

Figure 3.22 shows how the STM and and lock versions of WormBench scale. According to these results the STM version of WormBench scales but it is much slower (even with 16 threads) than the single-threaded lock based execution of WormBench. Even with 16 threads the performance of STM is not comparable with the single threaded STM version. The reason for this is that the STM system incurs significant overheads when doing versioning for the accessed read and write set, especially on the case when the worms body and head is 8 (B[8.8]H[8.8]) and the transaction has big read and write set. Another issue that can be observed is that the performance of lock based version degrades when ran with more than 1 thread. The reason for this is that the Bartok runtime is optimized for the case when the "lock" operation targets a lock that is not held. If the "lock" operation finds that the runtime lock has been already set by an earlier compare-and-swap operation then an OS mutex is created and thread blocked. In our case WormBench uses global lock which is most likely acquired and this way reflected negatively to the total throughput.

Figure 3.23 summarizes the relationship between the through-put (total number of moves per millisecond), the body length and head size, and the BenchWorld size. From the different charts (a), (b), (c) and (d) altogether is interesting to note here that the increase in the body length and head size have significant impact on the throughput. The obvious reason for this is that when the body length and head size becomes larger (especially head size, which has a O(n2) impact)

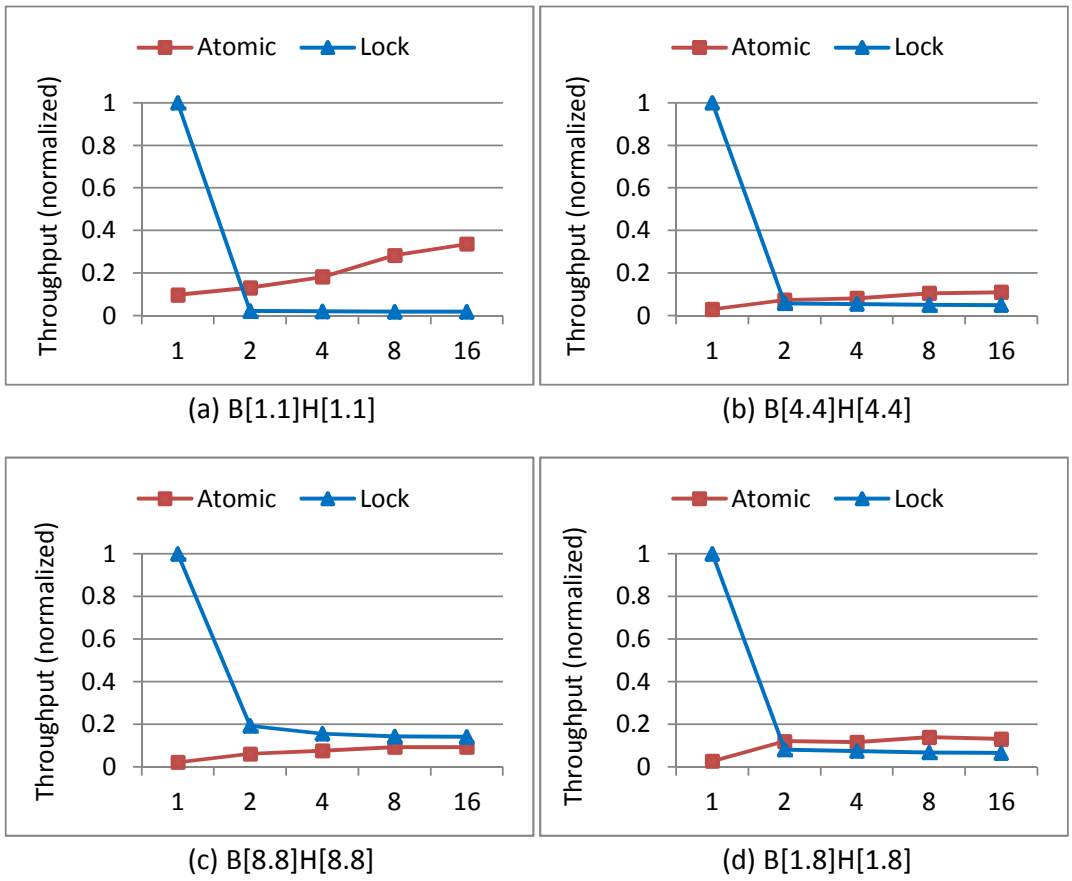


Figure 3.22: Comparing the performance between lock based synchronization and transactional memory synchronization (higher values are better). (a) all worms have body length and head size 1; (b) all worms have body length and head size 4; (c) all worms have body length and head size 8; (d) both the body length and head size of every worm is randomly selected from the range [1, 8].

the input to the worm operations become larger and they spend more time doing computation. For example, in the case with a head size of 1 summing has only one node to add but in the case with head size of 8 has 64 nodes. Another reason is that when body length and head size increase, transactions become larger and their working set increases super linearly. The overhead for maintaining big read and write sets along with the increased probability for aborts becomes higher. This can be better seen in Figure 3.22-(c) with B[8.8]H[8.8], when the transactional version of WormBench always performs worse because of the overhead incurred by the versioning and frequent aborts.

Figure 3.24 shows the average number of the objects opened for read or write per transaction. The *unfiltered* read set and write set (denoted as *UfR* and *UfW*) represent all the objects to which the TM system attempted to access and the filtered read and write set (denoted as FR and FW) represents the actual number of objects versioned by the TM systems. For example, it may happen that one object or memory location is once versioned and later accessed again. In this case the TM system filters it and does not allocate an entry for the second access. In Figure 3.24 is interesting to see that although the unfiltered read and write set increases for the different sizes of the worms, the filtered set remains constant.

Figure 3.25 shows the rate of successful commits (opposite to aborts). The commit rate in all the run configuration is very high. One reason for this is mainly because of using big BenchWorlds. Based on the results in this graph, we can conclude our previous observation: since the commit rate is high, the primary factor affecting the performance of B[8.8]H[8.8] configuration is the versioning overhead.

Figure 3.26 shows the commit rate results of run configuration with worms initialized for BenchWorld with size 128x128 and used in BenchWorlds with larger sizes (see Figure 3.21). The results in this figure are different from Figure 3.25 since its purpose is to show how the initialization of the worms affect the commit rate. The obtained results does not significantly differ from those in Figure 3.25 because we initialized the worms with big worm operations streams. Consequently, this long execution has effectively decreased the impact of the conflicts occurred at the beginning of the execution when the worms were relatively closer to each other. This configuration can model a TM-execution which has phases:

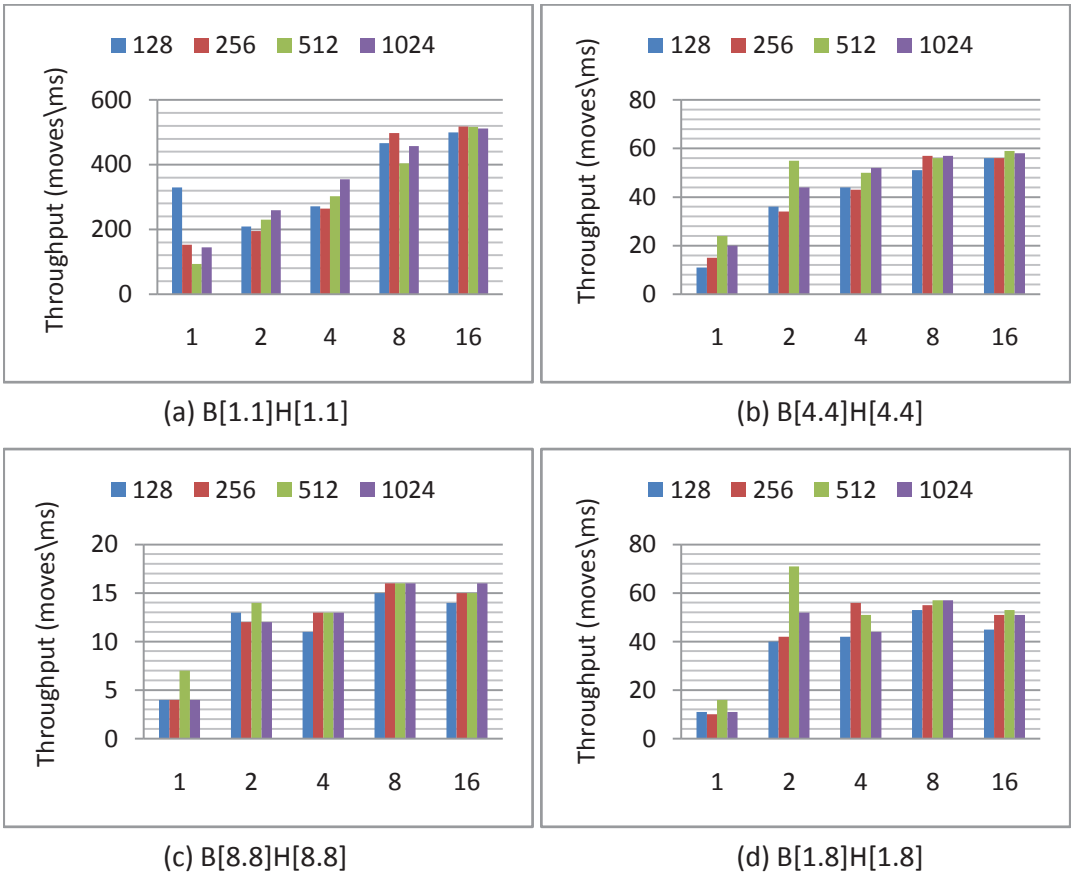


Figure 3.23: Relationship between throughput (total number moves per millisecond), BenchWorld size and the worm's body length and head size (higher values are better). (a) all worms have body length and head size 1; (b) all worms have body length and head size 4; (c) all worms have body length and head size 8; (d) both the body length and head size of every worm is randomly selected from the range [1, 8].

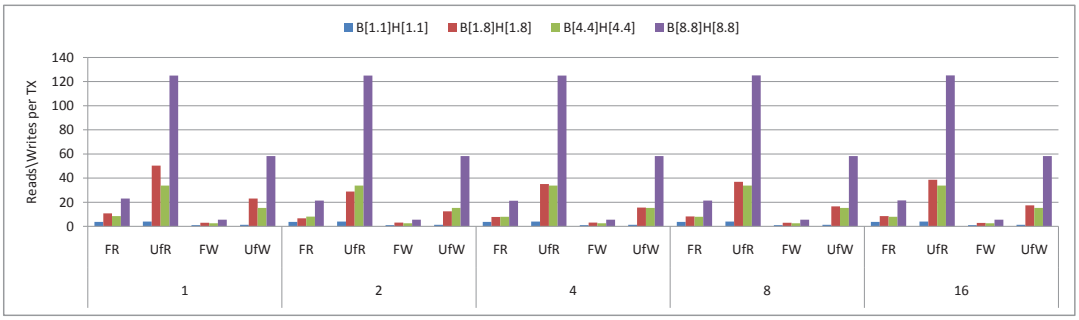


Figure 3.24: The number of unfiltered reads (UfR) and writes (UfW) per transaction and the number of filtered reads (FR) and writes (FW) per transaction.

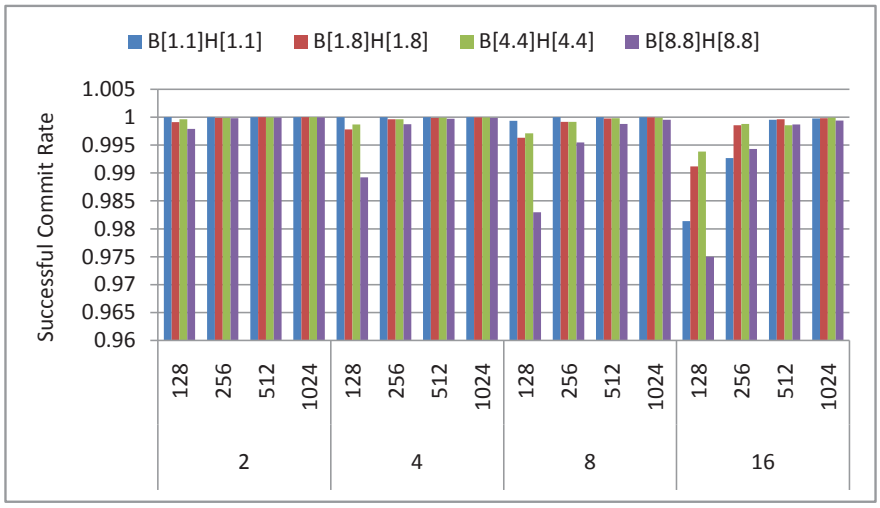


Figure 3.25: The average commit rate for all configurations. We omit the case for 1 worm (thread) because it is always 1.

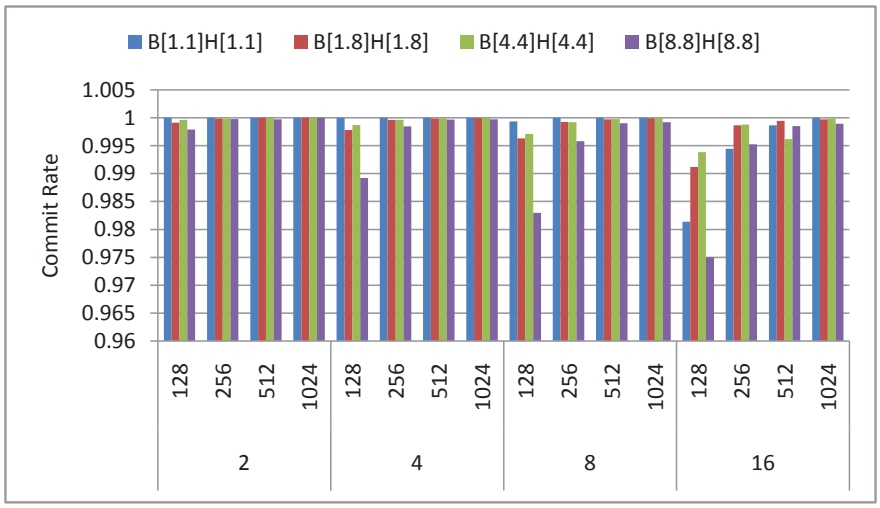


Figure 3.26: The commit rate when worms are initialized for BenchWorld with size 128x128 and then used in larger BenchWorlds - 128x128, 256x256, 512x512 and 1024x1024.

in the first phase it starts with a high conflict rate and continues with a lower conflict rate in the second phase. This characteristic of WormBench could be very useful in testing how well adaptive TM systems perform in the presence of changes in runtime TM-application behavior.

Based on the analyzed results in this section and the described characteristics of WormBench in the previous section, we will next show by example run configuration that WormBench can mimic the behavior of genome application from STAMP.

3.4.5 Modeling a TM Application

To demonstrate that WormBench is highly configurable we prepared a run configuration that has the similar transactional characteristics of the genome application from the STAMP benchmark. Table 3.7 compares the TM and runtime characteristics of the *genome* (Gen.) application and the run configuration for WormBench (WB) that mimics genome. *Read per TX* is the reads and *Write per TX* is the Writes. The commit rate and the number of reads (R) is very similar to the original values in genome. The proposed run configuration scales

up closely following the speedup rate of the original application. The number of writes (W) per transaction in WormBench is a little bit higher than in the original application but a careful tuning would be possible to lower writes and at the same time keep the other parameters unchanged.

	Commit Rate		Read per TX		Write per TX		Speedup	
T#	Gen.	WB	Gen.	WB	Gen.	WB	Gen.	WB
1	1	1	36.36	31.48	1.37	1.96	1	1
2	0.998	0.998	34.26	31.60	1.37	1.96	2.18	1.4
4	0.994	0.995	37.97	31.81	1.37	1.96	3.47	2.2
8	0.985	0.987	46.219	32.30	1.37	1.96	5.43	2.87

Table 3.7: Modeling Genome application with WormBench.

To obtain the results shown on Table 3.7 we used the following run configuration:

- Worms body length = 1
- Worms head size = 4
- BenchWorld of size 52x52
- Randomly generated stream of worm operations, where the ration between the worm operations was- Operations(1:2:3:4:5:6:7:8:9:10:11:12:13:14:15) = Ration(1:1:1:0:0:2:1:1:1:1:1:1:2:0:0)

This is just a small example that demonstrates the high configurability of WormBench and how it can be used to reproduce the runtime and TM characteristics of a specific multi threaded application.

3.5 Porting STAMP

To test extensions in the Bartok STM Bayes, Genome and Intruder from the STAMP TM benchmark suite [20] were ported from C to C#. The C versions were implemented in a modular object oriented style and it was straightforward to port them to C#. In C# `atomic` blocks were annotated with the available language

construct supported by the Bartok compiler. In the original STAMP applications, the memory accesses inside `atomic` blocks are made through explicit calls to the STM library, whereas in C# the calls to the STM library are automatically generated by the compiler.

After porting these applications had performance problems mainly due to the object granularity of conflict detection which Bartok-STM supports. After series of optimizations which are described later in Chapter 6 the performance of these applications was improved and aligned with their original versions.

3.6 Summary

In summary, the experience of using `atomic` blocks and TM in large applications such as AtomicQuake showed that indeed TM makes parallel programming easier than locks. TM is found to be useful for implementing fine grain synchronization for high number of objects or pointer based data structures such as trees and graphs. TM simplifies parallel programming in two ways. First, the programmer does not need to know which data is shared. In large programs it could be very difficult to identify all sides where a data can be accessed by two or more threads, particularly if the data is pointer based. Second, the programmer does not need to associate locks with the shared data and keep document this for future maintenance. Third, the programmer does not need to manually implement atomicity using locks for the operations which mutate shared data. Forth, the programmer does not need to use any locking schemes to avoid deadlocks. All these operations are transparently handled by the underlying TM. Besides synchronization, another place where TM can provide a gentle solution is failure atomicity. In AtomicQuake was demonstrated how TM can be used to recover from an error which in other case would cause the process to terminate.

Performance results obtained from AtomicQuake were encouraging. The STM version of AtomicQuake scaled well. However, compared to the lock-based version STM had high single threaded overhead ranging between x4-x5 time slow down. One reason for this could be that the compiler optimizations did not filter the operations on non-shared memory and they were also indirected through

the STM library. This would suggest that more work should be done on effectively distinguishing the operations on shared and non-shared memory. Another, performance problem of using STM was the sudden performance degradation on high number of threads. One reason for this could be the use of unnecessary large `atomic` blocks or presence of false conflicts. Unfortunately this problem left unstudied because the tools that were used were proprietary and did not provide sufficient profiling information.

Also, the experience of using `atomic` blocks and TM in a large application showed that TM is not yet mature technology for developing production stable software. It was encountered that more work should be done on semantics of transactions, language extensions, compiler integration, debuggers and profilers.

AtomicQuake is a rich TM workload which can be used to benchmark complete TM implementations. Its `atomic` blocks have different transactional characteristics. Inside `atomic` blocks there are uses of IO operations, error handling, nesting, composition and function calls.

WormBench is another TM application. Unlike AtomicQuake, it is parameterized synthetic workload. Its development did not have the goal of studying how `atomic` blocks and TM can be used but rather to deliver a tool that can be easily configured to have specific runtime characteristics.

In this dissertation, STAMP applications were used to evaluate the extensions made on the Bartok-STM compiler which are described in the remaining of the theses. They were also used to demonstrate series of optimization techniques for TM. Because of their small code base they were a good choice to port from C to C#.

Chapter 4

Debugging

Many researchers have developed research prototypes for `atomic` blocks, some based on static analysis for automatic lock inference, and others based on various kinds of transactional memory (TM), either implemented in software (STM) or hardware (HTM) [56]. However, based on the experience in our team of developing complex transactional applications such as AtomicQuake [135] (Section 3.3), QuakeTM [42], RMS-TM [67], WormBench [134] (Section 3.4) and Haskell-STM [52] we found it difficult and frustrating to use current debuggers when writing programs using `atomic` blocks and TM. This experience has motivated us to study how to extend debuggers to better support transactional applications.

This chapter presents the new principles and approaches for debugging transactional programs that were developed. In particular, we introduce the idea of distinguishing between debugging at the level of `atomic` blocks, and debugging at the level of transactional memory. When working at the level of `atomic` blocks, the programmer should only be aware that the blocks run atomically and in isolation: the programmer should not see implementation details such as exactly how `atomic` blocks are built over TM, or the internal algorithms used by a given TM implementation. Thus, when a breakpoint fires in an `atomic` block, the interrupted thread will be the only one in any `atomic` block. If the programmer single steps through the block, they will not see conflicts, transaction re-executions, and so on. A rule of thumb is that, at this level, the experience using the debugger

should be the same, whether `atomic` blocks are built over TM, or whether they are built over a static analysis for lock inference.

Conversely, when debugging at the lower level of transactions, the programmer is presented with a view to the underlying implementation of `atomic` blocks i.e. TM. This view is intended for debugging performance errors—for instance, identifying the instructions that are responsible for conflicts between transactions. Transactions represent the runtime execution of `atomic` blocks and have various attributes such as the number of aborts, status, priority, nesting level, and read and write sets. This information is helpful in debugging pathological cases such as forms of starvation [18]. In addition, besides finding errors, the debugger must be extended to handle basic information about the transactions, such as the read and write sets, in order to present the user with a correct view of memory. For example, in lazy versioning STMs that buffer the updates until commit, the user might be confused if the values of the variables in a watch list do not change while stepping inside an `atomic` block. Moreover, the user might be interested in debugging inside a particular `atomic` block only when a specific change in its state happens such as a transition from valid to invalid. To help in these situations, the user can additionally use the debugger to monitor for various events associated with the change of the transaction status and when for example a conflict is detected, the debugger will break automatically and display relevant information such as conflicting threads, statements and memory addresses.

New debugger abstractions enable the control of transaction and their state dynamically. These abstractions provide mechanisms to create and to remove *debug-time transactions* under the control of the debugger without changing and recompiling the source code. Such features are useful when investigating errors such as data races, atomicity violations and order violations – much as existing debuggers provide abstractions for modifying the contents of data in memory when investigating errors.

This chapter continues in Section 4.1 with a discussion on the problems which motivated the study on extending debuggers with support for `atomic` blocks and TM. Section 4.2 subways the state of the art in the area and relates this work to the others. Section 4.3 describes the design of the debugger framework and its implementation on WinDbg and Bartok-STM. Section 4.4 introduces the

approach of debugging applications at the level of `atomic` blocks. Section 4.5 introduces the approach of debugging applications at the level of transactions. Section 4.6 introduces the new debugger abstractions for managing transactions at debug-time. Section 4.7 summarizes this chapter.

4.1 Motivation

Extending the debuggers with support for transactional applications was primarily motivated by the experience of developing AtomicQuake [135] (see Section 3.3). While developing AtomicQuake there were encountered several difficulties of debugging this application. These difficulties are:

1. While stepping in the code the debugger always steps inside the `atomic` block instead of executing it as if a single instruction. This makes difficult to finding synchronization errors which manifested at the level of `atomic` blocks i.e. atomicity violations.

2. It is very difficult to debug wrong code inside the `atomic` because: 1) the debugger does not distinguish between speculative and non-speculative values and 2) the sudden aborts return the control flow to the beginning of the `atomic` block. The latter is particularly frustrating when debugging the implementation of function called inside `atomic` blocks.

3. The debugger does not have abstractions to see and modify the state of the TM system such as removing/inserting entries to the read or write set.

4. The debugger does not have abstractions to break when the TM state changes, for example to break when conflict happens and show relevant information about the conflict. Such functionality is useful to examine specific conflicts which hurt the performance.

5. The debugger does not have abstractions to: 1) create new `atomic`, 2) remove existing `atomic` blocks, 3) change the scope of existing `atomic` blocks without exiting the debugging sessions. Such functionality is particularly

useful for identifying non-deterministic synchronization without exiting the debugging session.

6. The debugger does not have mechanisms to abort or explicitly commit a transaction.

These problems are explored and addressed with the work described in this chapter.

4.2 Related Work

In a parallel work with this one, Herlihy and Lev have developed an infrastructure for debugging transactional applications—tm_db [59]. From a user's perspective, compared to our work, when debugging a transactional application with the abstractions that Herlihy and Lev introduce, it will look like debugging at the level of transactions (discussed in Section 4.5). Their approach has the objective to properly integrate the debugger with the TM implementation. The primary focus of tm_db is to consistently expose the TM state through the debugger without changing the existing debugging conventions. In addition to transaction-level debugging we introduce the notion of debugging at the level of `atomic` blocks, attempting to abstract over whether or not these are implemented with TM. We also propose and implement mechanisms to create debug time transactions, split `atomic` blocks and modify the state of transactions under the control of the debugger. In tm_db Herlihy and Lev introduce important concepts such as logical value, scopes, distinction between transactional reads, writes and their respective conflict coverages. These new concepts abstract the internal organization of different STM systems. Logical values are necessary for preserving the isolation property of transactions when debugging at the level of transactions. Abstracting the reads and writes with their respective coverages hides the internal mechanism to manage the read and write sets and also help in identifying false conflicts. Incorporating these new abstractions into our extension would provide users an uniform view to the TM state when debugging at the level of transactions.

Using their debugging infrastructure, Herlihy and Lev provided support for 8 different TM implementations [74]. To do so, they implemented separate Remote Debugging Module (RDM) systems, one for each library variation, and they extended the STM libraries with support for debugging.

In earlier work, before tm_db, Lev and Moir discussed how the debugger and the TM implantation should by integrated [73]. They surveyed features that a debugger could provide by leveraging the underlying TM system. From their work, we were inspired that seeing the read set and write set of transactions can help to understand the reason for aborts.

Chafi *et al.* have developed a micro architectural extension TAPE [23] for the Transactional Coherence and Consistency [24] system that has HTM support. They used TAPE to profile and optimize transactional applications by studying the locations where transactions conflict much like we do in conflict point discovery but in STM. These two approaches can be combined in a hybrid transactional memory system.

Recent work carried by Gupta *et al.* leveraged the existing infrastructure in a hardware transactional memory system RaceTM [47] to detect data races in multi-threaded applications. The combination of this functionality and debug-time transactions would be a complete tool to find and fix data races in multi-threaded applications at debug time.

4.3 Design and Implementation

We prototyped our ideas in an extension module for the publicly-available WinDbg debugger [84]. Concretely, we target transactional C# applications compiled with Bartok [55] compiler. However, our design decisions are motivated by maintaining applicability of our approaches to other debuggers, other TMs, and to non-TM implementations of `atomic` blocks.

WinDbg is a multi-purpose debugger for Win32 applications. Its functionality can be extended by using the Microsoft Debug Engine Extension APIs [83]. WinDbg extensions are Dynamic Link Libraries (DLL) that implement and export a number of callback functions. Some of these callbacks are required by the

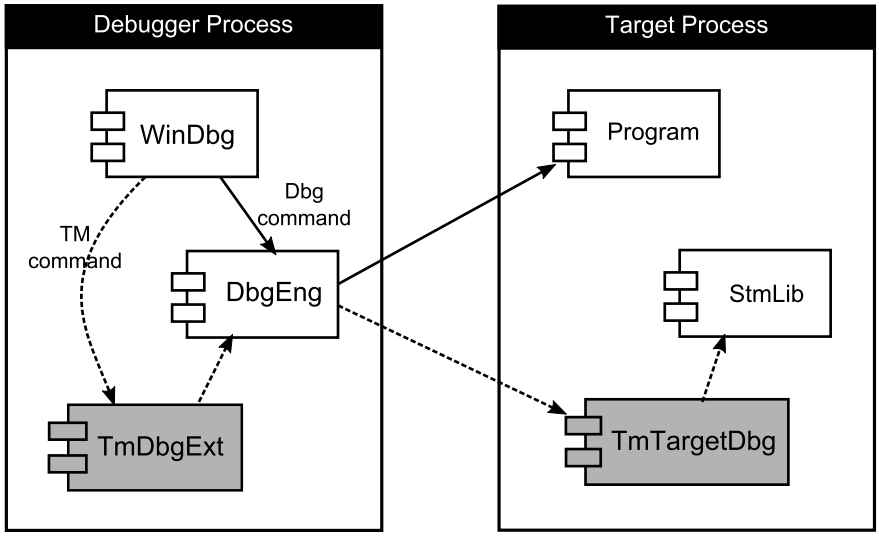


Figure 4.1: Decoupled design approach for the debugger extension. The components in gray represent our extension and the dashed lines represent TM operations. The implementation of TmTargetDbg is specific for our STM library and the implementation of TmDbgExt is specific for WinDbg family of debuggers.

debugger for the extension's integration, and other callbacks implement the additional user commands that extend the debugger functionality or let it visualize specific data structures.

Bartok is an ahead-of-time C# compiler with language level support for `atomic` blocks. The runtime execution of the `atomic` blocks in applications compiled with Bartok is handled by an STM library which from now on we will refer to as Bartok-STM. Bartok-STM updates memory locations in-place by logging the old value for rollback in case a conflict happens. It detects conflicts at an object granularity, eagerly for write operations and lazily for read operations.

In the following sections we introduce our design and implementation of the debugger extension and then from Section 4.4 return to the high-level debugging approaches.

4.3.1 Design Approach

We have chosen a decoupled design for extending WinDbg. Our design consists of two components: a debugger extension library (TmDbgExt) and an STM-library debug helper (TmTargetDbg). Figure 4.1 shows the structure of the system. TmDbgExt implements the end-user debugger commands for use with `atomic` blocks and transactions. It is dynamically loaded by WinDbg and runs as part of the debugger process and uses the debugger engine (DbgEng) to access the target. TmDbgExt is specific to a particular debugger, but independent of the TM in use. Conversely, TmTargetDbg runs in the address space of the program being debugged. TmTargetDbg is specific to the TM, but independent of the debugger.

We were inspired by the approach described in Lev's presentation [71]. Comparing with Herlihy and Lev's subsequent paper [59], we have only one component at the debugger side (TmDbgExt), whereas Lev's design uses two (*tm_db* and a *Remote Debugging Module*, RDM). tm_db defines an common interface for implementing extensions to debug transactional applications. It can be used with all debuggers providing the `proc_service` interface and is independent of the TM implementation. RDM provides tm_db with functionality for debugging a particular TM. Within the target process, the TM runtime system provides a support layer (RTDB). We chose to avoid placing any TM-specific components on the debugger side—the developer of our TmTargetDbg will not need to know about the debugger and vice versa. Ultimately, we might be tempted to define a common interface and communication mechanisms between TmDbgExt and TmTargetDbg—but this seems premature at the moment.

We also experimented with an alternative approach which implements all the functionality in the debugger extension (TmDbgExt), without the helper component in the target process. In this approach the debugger extension is coupled with the STM library implementation and depends on the layout of the data structures, size of buffers, alignment, and so on. For instance, suppose that we want to check if a specific memory address is in the read set of a transaction. The debugger-side module would need to be coupled to the layout of the data structure representing the read set entry and the field where the address is stored.

Also, the module has to know any possible alignment restrictions that the compiler might apply. Modifying the read set entry data structure by adding a new field or compiling for different architecture (e.g., 64-bit) would require changing and re-testing the debugger extension. We felt that this model was not a good fit with rapidly-evolving transactional memory systems.

We believe our decoupled design approach can readily be applied to implementations of `atomic` blocks over other TMs; the details of TmTargetDbg will vary, depending on the exact data structures used, but the approach will remain the same.

4.3.2 Interaction Between TmDbgExt and TmTargetDbg

The interaction between the debugger and the STM library has two levels of indirection. First, TmDbgExt accesses TmTargetDbg over the debugger engine API and then TmTargetDbg accesses the STM internals (see Figure 4.1). TmTargetDbg acts as a wrapper for the STM library and exports a set of functions listed in Table 4.1. TmDbgExt may query or modify the STM state by setting a call to one or more of these functions. To safely execute a function in the target process, TmDbgExt saves the process context prior the call and restores it after the call. For simplicity, we have designed the prototypes of the TmTargetDbg functions in a way that if the return value is larger than a register (e.g., an array or a data structure) the value is stored in a temporary location and the address to this location is returned.

4.3.3 Internal Breakpoints

We use breakpoints to implement many of our new debugger features. For instance, when debugging at the level of `atomic` blocks and a normal breakpoint fires inside an `atomic` block, we must check that the current transaction is valid, and then "clean" the visible state of other threads (e.g., by rolling back transactions that other threads are in). This provides the impression of isolation. In many examples like this we either need to cause the target process to execute STM-helper functions, or we need to roll forward application code in the target process.

Operation	*Description*
GetTxStatus	Get the status of the transaction.
SetTxStatus	Set the status of the transaction.
GetPriority	Get the priority of the transaction.
SetPriority	Set the priority of the transaction.
GetReadSet	Get the read set of a transaction.
GetWirteSet	Get the write set of a transaction.
AddToReadSet	Add entry to the read set.
RemoveFromReadSet	Remove an entry from the read set.
AddToWriteSet	Add entry to the write set.
RemoveFromWriteSet	Remove an write from the read set.
GetNestingLevel	Get the nesting level of a transaction.
GetOriginalValue	Get value before a speculative update.
GetSpeculativeValue	Get value after speculative update.
IsTxIrrevocable	Check if a transaction is irrevocable
SwitchToIrrevocable	Switches transaction to irrevocable mode.
StartIrrevocableTx	Starts a transaction in irrevocable mode.
CommitIrrevocableTx	Commits an irrevocable transaction.
SplitTx	Splits a transaction

Table 4.1: The API of TmTargetDbg component.

Both of these operations involve adding temporary breakpoints in addition to those set by the user (e.g., we must regain control after rolling forward). We refer to these as "internal" breakpoints. As described later in this thesis, we used internal breakpoints to override the step command to interpret `atomic` blocks as a single statement (Section 4.4.1), to implement watchpoints (Section 4.5.1), to implement debug-time transactions (Section 4.6.1) and to split `atomic` blocks (Section 4.6.2).

We use a breakpoint-time callback to distinguish ordinary user-breakpoints from internal breakpoints. The callback overrides the default debugger behavior of suspending the target program when an internal breakpoint is hit and, if necessary, it executes complementary actions associated with the internal breakpoint.

The diagram in Figure 4.2 shows how the callback works. When a breakpoint is hit, the callback checks whether it is a normal breakpoint, or an internal breakpoint. If it is a normal breakpoint and the event thread is executing a transaction, the callback executes a complementary action to switch the transaction to irrevocable mode [115; 126] and breaks to the debugger prompt (Section 4.4). If the breakpoint is internal, the callback executes a complementary action based on its type (purpose). Also, depending on the type of the internal breakpoint the debugger may either break or continue execution as if the breakpoint is not hit.

4.3.4 Probe Effect and Overhead

Of course, as with many debugging techniques, the approach described here might add a probe effect because of the changes on the underlying STM. We have measured the probe effect over the Red Black Tree micro benchmark which reflects even the minimal overheads in an amplified scale. Table 4.2 shows relative difference between a binary compiled without any additional logging and binary compiled with the logging required for conflict point detection. Column *Execution Time* shows the relative difference between execution times and column *Aborts* shows the relative difference between abort rates. This experiment suggests that while the probe effect may change the fine-grain behavior of the program it does not introduce or remove high level contention. Qualitatively, when reducing contention on hot spots identified by conflict point discovery, contention

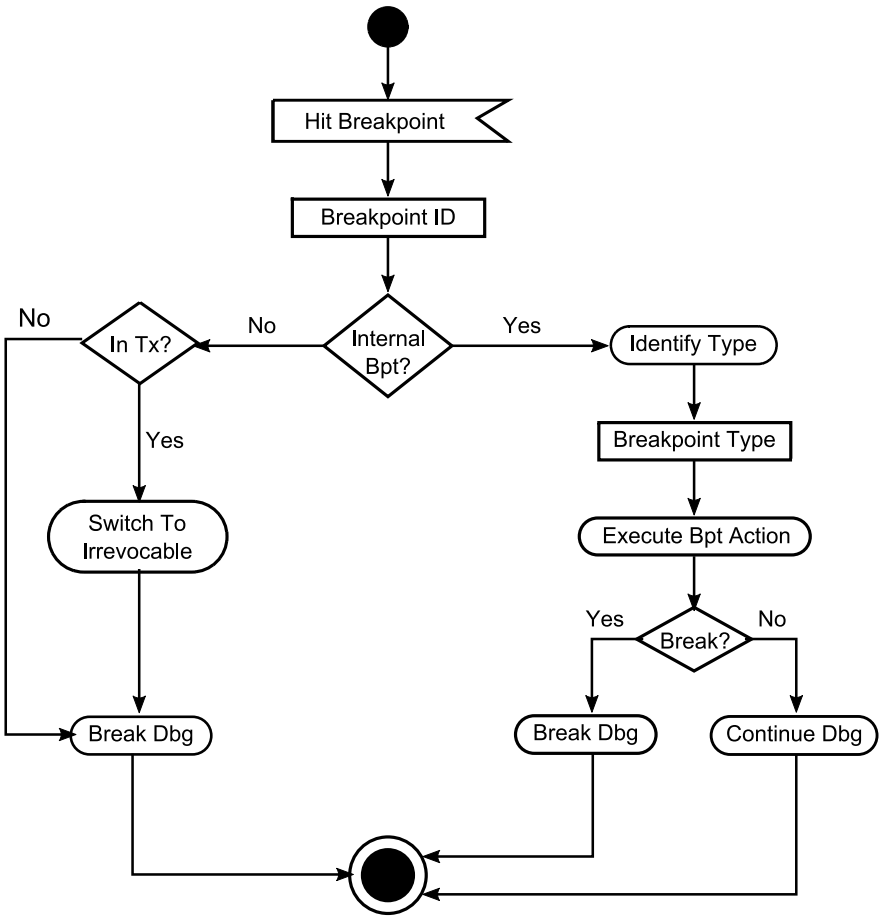


Figure 4.2: Using a breakpoint callback to distinguish between normal user breakpoints and the internal breakpoints. Also, the breakpoints fired during transaction execution may require to do complementary actions such as switching the transaction to irrevocable mode.

Threads	Execution Time	Aborts
1	0.0%	n/a
2	0.4%	1.2%
4	6.0%	4.5%
8	10.6%	1.6%
16	5.6%	10.0%

Table 4.2: The probe effect of the debugger extensions. In this experiment we used the Red Black Tree microbenchmark.

in the underlying program is reduced. Similarly, programs with low contention under normal execution have low contention under conflict point discovery.

4.4 Debugging at the Level of Atomic Blocks

This section discusses the approach for debugging transactional applications at the level of `atomic` blocks. We extend the debugger to model the semantics of `atomic` blocks, presenting the user with the impression that they run with atomicity and isolation (even when the underlying implementation uses TM).

Consequently, when debugging a program using `atomic` blocks, we provide facilities to single-step over entire blocks so that they appear as indivisible operations (Section 4.4.1), and to step into a block while preserving the appearance that it is executing in isolation (Section 4.4.2).

By analogy, a debugger for a language implemented with garbage collection (GC) will abstract away the details of how the heap is structured—e.g., when single-stepping, it would not step into the GC implementation if it runs, and it would clear and re-set data watchpoints if the underlying objects are relocated.

4.4.1 Stepping Over Atomic Blocks

The atomicity property of `atomic` blocks guarantees that the statements comprising the `atomic` block execute either all or none. When debugging higher-level concurrency errors in transactional applications, the user may therefore have the

```
 1: atomic {
 2:    ...// Initialize the bounding box
 3:    if (ent->v.modelindex)
 4:       SV_FindTouchedLeafs (ent, sv.worldmodel->nodes);
 5:    ent->num_leafs = 0;
 6:    if (ent->v.modelindex)
 7:       SV_FindTouchedLeafs (ent, sv.worldmodel->nodes);
 8:    if (ent->v.solid != SOLID_NOT) {
 9:       tm_block_flag = true;
10:       i=1;
11:       node = sv_areanodes; // Areanode tree
12:       while (1) {
13:          if (node->axis == -1)
14:             break;
15:         if (ent->v.absmin[node->axis] > node->dist) {
16:             node = node->children[0];
17:             i *= 2;
18:          }
19:          else if (ent->v.absmax[node->axis] < node->dist) {
20:             node = node->children[1];
21:             i = i*2 + 1;
22:          }
23:          else
24:             break;
25:       }
26:       if (ent->v.solid == SOLID_TRIGGER)
27:          InsertLinkBefore (&ent->area, &node->trigger_edicts);
28:       else
29:          InsertLinkBefore (&ent->area, &node->solid_edicts);
30:    }
31: } // end atomic
```

Figure 4.3: The body of the `atomic` block in function `SV_LinkEdict` from Atomic Quake which is responsible for changing the location of an object such as a player from its old to the new position in the map (areanode tree).

expectation that the debugger will execute the `atomic` block in its entirety without being interested in what is going on inside—much as the user may step over a complete function call.

Earlier work that studied the construction of parallel programs with `atomic` blocks and TM [92; 100] and our experience of developing such applications [42; 67; 93; 134; 135] suggests that programmers organize transactional synchronization between threads in a different, more abstract, way by relying on the atomicity of complete transactions but not identifying the individual shared data structures to protect them with locks. In this approach, the concurrency errors in transactional applications are coarser and manifest on the level of `atomic` blocks and not on the level of individual statements inside the `atomic` block.

Existing debuggers are not aware of `atomic` block boundaries and so they do not provide the illusion of atomicity. In such a case, instead of helping to identify the concurrency problem, the debugger may cause additional confusion, especially if the `atomic` block contains sophisticated logic and function calls. For example, Figure 4.3 shows the body of the `atomic` block in function `SV_LinkEdict` taken from the AtomicQuake code (see Section 3.3). This function is responsible for changing the location of a game object (e.g., a player) from one to another location in the game map. Suppose that we are searching for an error and want to see the state of the map data structure (i.e. `sv_areanodes` line 11) before and after executing the `atomic` block. When we advance in the debugger, we would normally proceed by stepping into each of the statements inside the `atomic` block. This will show the intermediate changes, rather than the overall effect of the block. Furthermore, if the transaction implementing the `atomic` block aborts part-way through, the user may find execution back at the start of the first statement.

Without debugger support for TM, a workaround for this problem is to put a breakpoint at the end of the `atomic` block (i.e. line 31) and to continue execution up to that point. This has the effect of executing the `atomic` block as a single statement.

To support execution of complete `atomic` blocks, we provide a distinct `tmstep` operation. This steps over the whole `atomic` block in a single operation. To implement this, TmDbgExt puts internal breakpoints at the functions exported by TmTargetDbg that are called at the start and end of an outermost transaction

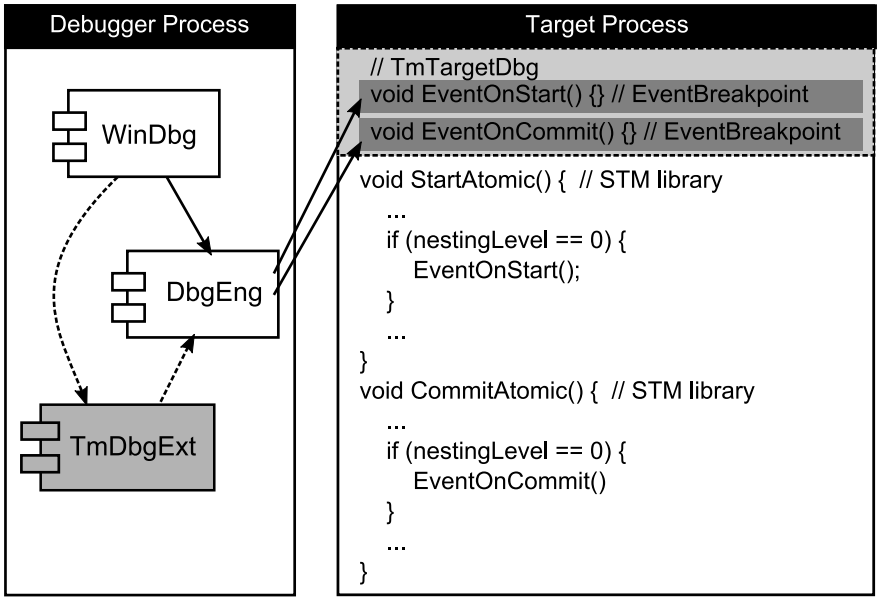


Figure 4.4: The illusion of atomicity in TmDbgExt is implemented by putting internal breakpoints at the functions `EventOnStart` and `EventOnCommit` called by the STM library when outermost transactions start and commit respectively. When the breakpoint on function `EventOnStart` is hit, TmDbgExt continues execution in `go` mode, and when the breakpoint on `EventOnCommit` is hit TmDbgExt restores the execution to `step` mode.

(Figure 4.4). These breakpoints are enabled by default and, when the first one is hit upon starting a transaction, the debugger continues to execute until it reaches the matching commit function. When committing the outermost function the breakpoint on the commit function is hit and this time the debugger switches back to normal stepping mode.

4.4.2 Stepping Inside Atomic Blocks

The isolation property of `atomic` blocks guarantees that threads will not see the intermediate updates made by a thread which executes an `atomic` block. Consequently, we provide a mechanism to preserve isolation when stepping into `atomic` blocks.

This is intended for debugging errors within a single `atomic` block—for in-

stance, if our code in function `InsertLinkBefore` (Figure 4.3 line 26–29) is wrong, and its internal logic needs to be examined. Debugging within an `atomic` block is activated automatically when a breakpoint is hit while executing a transaction. For example, if the user puts a breakpoint at line 27 in Figure 4.4 then the user will be able to advance inside the `atomic` block by stepping over each statement.

To preserve the appearance of isolation, we must take care to prevent interference between transactions—e.g., consuming speculative updates from concurrent transactions, operating on an inconsistent view of memory, or being aborted and re-executed. For instance, in the code example from Figure 4.3 the root of the areanode tree is assigned to a local variable (line 11), and if a second transaction commits a change to the root, then `InsertLinkEdict` might operate on invalid data. Debugging logic inside an `atomic` block based on invalid values but not yet detected conflict, camouflages the actual problem and violates isolation.

We preserve isolation by switching the transaction being debugged into irrevocable mode [115; 126] (i.e., a transaction that is guaranteed to commit). Our implementation of irrevocable transactions is simplistic: before switching to irrevocable mode the TM library validates all transactions and makes sure that the only transaction being executed is the irrevocable one (rolling back any others). Thus, while stepping through an `atomic` block, the user will see only actual values and never see transactional aborts.

If a conditional breakpoint is reached while executing a transaction, we first validate the transaction, and if the validation passes successfully we break into the debugger. If validation fails, then the transaction is aborted and re-executed, without breaking into the debugger.

This is necessary to prevent invalid transactions from falsely suspending the execution, and reflects our intended semantics for `atomic` blocks which are designed to abstract the details of particular TM implementations.

4.5 Debugging at the Level of Transactions

When debugging at the level of transactions, the debugger extension deliberately exposes a TM-based implementation of `atomic` blocks. The aim is to provide the user with means to discover and reason about pathological situations, such

as those described by Jayaram *et al.* [18]. Such examples can harm overall performance or prevent progress.

When debugging at the level of transactions, the user can step into the statements inside an `atomic` block without changing the transaction into irrevocable mode like we did in Section 4.4. In such a case, when advancing line-by-line over the source code, the execution of two or more `atomic` blocks may be interleaved, and the user may observe the effect of this interleaving on the TM system. At any time, the user can see the state of any active transaction, and inspect the following attributes:

- The status of the transaction such as valid, invalid, blocked.
- The priority of the transaction.
- How many times the transaction aborted and re-executed.
- The transaction's read and write set.
- Whether the transaction is irrevocable.
- The ID of the thread executing the transaction.
- The original and the speculative value of a variable.

The debugger must distinguish between original and speculative values in order to support some of its existing features. For example, a user might have a variable in a watch list that is speculatively updated in a transaction. The underlying value of this variable will not change in TM systems with lazy versioning (i.e., which buffer updates until commit). In such cases, the debugger must monitor transactional writes to check if this variable is updated by a transaction and display its most current value. Herlihy and Lev [59] make a more detailed analysis of this problem and discusses the changes for the current debuggers in order to support it.

By combining these primitive queries, we have also implemented richer operations to intersect the read or write sets of two or more transactions. This is intended to help the programmer understand the common data sets between

the transactions, and to discover pathological cases that prevent transactions to progress and hurt the overall application performance.

However, the user would usually prefer not to dive in the world of this complicated debugging which requires knowledge about the workings of the underlying TM implementation until a specific event happens such as a transition from a valid to an invalid state due to a conflict. We discuss transaction events in more detail in the next section.

4.5.1 Transaction Events

Our debugger extension can monitor transaction events that relate to changes in the status of the transaction and its read and write sets. The events that users can monitor are:

- Transaction start.
- Transaction commit.
- Transaction abort.
- New read or write set entry.

A user can set a watchpoint on any of these events. When the watchpoint is triggered, the debugger breaks and provides contextual information such as the event thread, the conflicting transactions, the conflict addresses, or the entry being added into the read or write set. To avoid interrupting the target process at uninteresting places, the user can also introduce filters for these events so that the event is triggered—for example, only if the conflict happens on a specific `atomic` block(s).

To be able to catch the transaction events as they happen, we define stub functions in TmTargetDbg for each of these events. The stubs are called by the STM library when the event happens. To break on an event, TmDbgExt places a internal breakpoint on the entry to the relevant stub. Also, to filter out irrelevant events, TmDbgExt can modify a *filter mask* variable defined in TmTargetDbg (see Figure 4.5). Depending on the filter criteria the STM library decides whether or

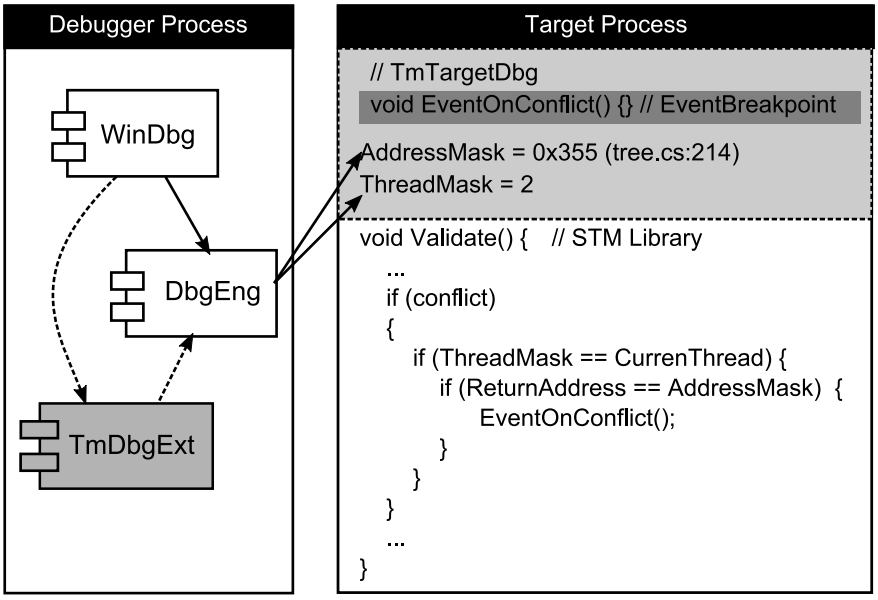


Figure 4.5: Filtering uninteresting events. The debugger extension sets filter mask for thread id 2 and instruction address to monitor for conflicts. When conflict happens the STM checks the masks and if they are true calls the function `EventOnConflict` which is set a breakpoint.

not to call the corresponding event function. We enable these tests only when compiling in debug mode.

4.6 Debug-Time Transaction Management

Our final set of debugger extension features allow the user to manage the transactions under the control of the debugger. At the level of `atomic` blocks, the user can create *debug-time transactions* or *split* `atomic` blocks. These features are intended for investigating errors in the source code, and trying to patch the errors without modifying and recompiling the source code (e.g., when testing out a hypothesis for what is causing a race condition).

Although it might be error prone, drawing analogy from current debuggers' functionality that allow users to modify the program aspects by changing the values of variables in memory or processor registers, we were motivated to implement operations that the user might use to change the state of the transactions

such as by adding or removing entries into the transaction's read and write set when debugging at the level of transactions.

4.6.1 Debug-Time Transactions

A *debug-time transaction* is a new debugger abstraction that helps for the correctness debugging of transactional applications. While debugging, a user may notice that `atomic` blocks are missing in certain places or that `atomic` blocks could be reduced in size. Figure 4.6 has a contrived example (line 26) where a data race occurs because an `atomic` block is too small. Figure 4.7 has an example where, instead of defining one large `atomic` block, the program uses two smaller blocks. In such cases, the user can create a debug-time transaction or enlarge the scope of an existing `atomic` block by marking the boundaries of the new `atomic` block on the source code. Thereafter, the debugger ensures that the debug-time transactions are executed atomically, as if regular `atomic` blocks, but without exiting the debug process to change and recompile the source code.

In Figure 4.8 we show a difficult to find atomicity violation example that we discovered in the QuakeTM [47] source code after a careful inspection. The error manifested in disconnecting the clients from the game session due to bad formatted messages. We checked the functions such as `WriteMulticast` which build these client messages and their definitions were all correctly synchronized. To see how the execution changes, we randomly created and removed temporary `atomic` blocks or coarsened existing ones. Due to the nondeterministic nature of the error, it took us quite long time to constrain the problematic location to the code that interprets Quake extension functions implemented in Quake C and compiled to intermediate representation. If we were able to create, remove and resize `atomic` blocks while debugging, we would find the problematic location easier. In this case we would save a lot of time from changing and recompiling the source code and trying to reproduce the error by re-establishing the client-server game session.

Later, by reverse engineering the Quake extension functions interpreted inside this problematic code, we noticed that there is one function (`FireAxe`) which calls

```
 1:  static public void Main(string[] args) {
 2:     Thread t1 = new Thread(ThreadEntryIncrement);
 3:     Thread t2 = new Thread(ThreadEntryDecrement);
 4:
 5:     t1.Start();
 6:     t2.Start();
 7:  }
 8:
 9:  static void ThreadEntryIncrement() {
10:     int temp = 0;
11:
12:     atomic {
13:         temp = counter;
14:         temp++;
15:         counter = temp;
16:     }
17:  }
18:
19:  static void ThreadEntryDecrement() {
20:     int temp = 0;
21:
22:     atomic {
23:         temp = counter;
24:         temp--;
25:     }
26:     counter = temp;
27:  }
```

Figure 4.6: An example where the `atomic` block in lines 22-25 is shorter and line 26 must be included in the `atomic` block.

```
   initially a = b = 0;
.
   Thread 1                    Thread 2
.
1:  atomic{
2:     a++;
3:  }                           atomic {
4:                                 a++;
5:  atomic{                     }
6:     b--;
7:     assert(a + b == 0);
8:  }
```

Figure 4.7: An example of incorrectly splitting a critical section in two smaller `atomic` blocks. The shown interleaving between thread 1 and thread 2 will result in violating the invariant that a+b=0.

```
   // Correctly synchronized function
   void
   WriteMulticast(message) {
     atomic {
       <update message buffer>;
     }
   }
.
       Thread 1                          Thread 2
1:  void FireAxe(){
2:     WriteMulticast(msg_part1);
3:                                       WriteCoordinate(coord);
4:     WriteMulticast(msg_part2);
5:  }
```

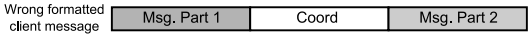


Figure 4.8: A difficult-to-discover atomicity violation from QuakeTM code. In a serial execution, the two calls to `WriteMulticast` function would be executed one after other and the two parts of the multicast message would be next to each other. To properly synchronize this is necessary to call `FireAxe` method inside an `atomic` block.

the function `WriteMulticast` several times to build the individual parts of a multicast message. This pattern of use is similar to calling `printf` to print multiple lines on the console. In a serial execution, these functions would execute one after the other and build a correct message. But in multi-threaded execution, although each `WriteMulticast` function is correctly synchronized a possible interleaving with another thread, like the one shown in the Figure 4.8, would result in a malformed packet.

The implementation of debug-time transactions, relies on the availability of irrevocable transactions in the STM library. When the user marks the start and the end of the transaction TmDbgExt gets the addresses of the statements using the debugger engine and puts internal breakpoints (see Section 4.3.3) at these places—one denoting the start and other denoting the end of the transaction. And when the start or end breakpoint is hit, TmDbgExt calls respectively the `StartIrrevocableTransaction` or `CommitIrrevocableTransaction` function from TmTargetDbg by following the method described in Section 4.3.2.

There is one more subtlety of calling function `StartIrrevocableTransaction`.

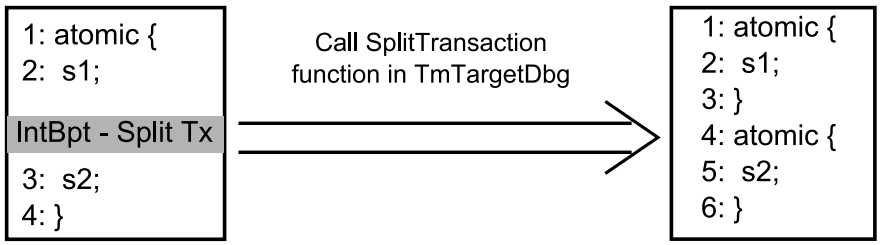


Figure 4.9: Splitting a transaction. TmDbgExt puts a internal breakpoint (IntBpt) denoting the place where the transaction is to be split. When the breakpoint is hit the debugger transparently calls a function `SplitTransaction` in the target process, creating the effect of committing a transaction and initiating a new one.

This method manipulates locks within the STM library and must synchronize with other threads (e.g., if they are also trying to start irrevocable transactions). Consequently, if we call this method by resuming only one thread in the target process and keep the other threads blocked may cause deadlock. Therefore, in this case we resume all target-process threads until the call to `StartIrrevocableTransaction` is complete.

4.6.2 Splitting Atomic Blocks

To split a large `atomic` block into two smaller ones, we provide the user with two alternatives. In the first alternative, while stepping inside an `atomic` block, the user can split the transaction for one time only at the place where the next statement is to be executed. In the second alternative, the user marks at which statement to split the transaction (see Figure 4.9). In the former case following the method for calling functions in the target process, described in Section 4.3.2, we call the `SplitTransaction` function from TmTargetDbg. In the latter case, TmDbgExt creates a internal breakpoint on the location where the transaction is to be split. Whenever any breakpoint is hit, TmDbgExt checks if it is used to split an `atomic` block and if so, the debugger transparently calls the `SplitTransaction` function and continues the execution without breaking into the debugger. In effect, function `SplitTransaction` commits the current transaction and then immediately initiates a new transaction.

One subtlety inherent to our STM implementation that should be considered implementing this feature is where the split point is introduced. The user should be disallowed to split atomic blocks in functions that are not defining the outermost transaction. In this situation, the second part of the transaction (e.g., lines 4-6) may not be able to roll back to an interior point (the place where the transaction was split) because the function stack is torn down.

We believe that users who want to optimize their transactional applications by decreasing the size of the coarse grain `atomic` blocks would greatly benefit from this feature. For example, at debug-time users can split the large `atomic` blocks and see how this affects the correctness and the runtime performance.

4.6.3 Modifying Transactional State

TmDbgExt implements user commands to directly modify the state of the transaction by changing any of its attributes and also adding or removing an entry into the transaction's read and write set while debugging at the level of transactions (see Section 4.5). All these operations may cause an incorrect execution of the application and it is the user's responsibility to use them reasonably. Adding an entry into the read or write set of a transaction may cause the transaction to become invalid and abort. The debugger extension detects such cases and warns the user by requesting to confirm the action. These operations are implemented by calling the respective functions from Table 4.1 which modify the STM state.

4.7 Summary

This chapter presented three different debugging approaches for transactional applications. Debugging at the level of `atomic` blocks provides users the same experience across different underlying implementation mechanism. The debugger is extended to reflect the atomicity and isolation properties of `atomic` blocks and this makes it easier to debug synchronization problems across different `atomic` blocks and incorrect code within `atomic` blocks. Debugging at the level of transactions assumes that the underlying implementation of `atomic` blocks is TM and exposes their typical attributes such as read and write set. Debugging by following

this approach is useful to discover pathological cases that have negative impact on the overall runtime performance. We introduced mechanisms for adding and removing `atomic` blocks under the control of the debugger which would make debugging synchronization problems such as atomicity violations easier. In our implementation of these features, we followed a general decoupled approach that can be applied to any debugger and TM system. The extensions that we made on the underlying STM system add marginal overhead and low probe effect.

Chapter 5

Profiling

Chapter 3 made conclusions that it is easier to write programs using transactional memory (TM). However, if a program is to perform well, then the programmer needs to understand which transactions are likely to conflict and to adapt their program to minimize this [18]. Several studies report that the initial versions of transactional applications can have very high abort rates [42; 92; 100]—anecdotally, programmers tend to focus on the correctness of the application by defining large transactions without appreciating the performance impact.

Various *ad hoc* techniques have been developed to investigate performance problems caused by TM. These techniques are typically based on adding special kinds of debugging code which execute non-transactionally, even when they are called from inside a transaction. This non-transactional debugging allows a program to record statistics about, for example, the number of times that a given transaction is attempted.

This chapter describes a series of methodical profiling techniques which aim to provide a way for a programmer to examine and correct performance problems of transactional applications. We focus, in particular, on performance problems caused by conflicts between transactions: conflicts are a problem for all TM systems, irrespective of whether the TM is implemented in hardware or software, or exactly which conflict detection mechanisms it uses.

In this work we follow two main principles. First, we want to report all results to the programmer in terms of constructs present in the source code (e.g., if an object X in the heap is subject to a conflict, then we should describe X in a way

that is meaningful to the programmer, rather than simply reporting the object's address). Second, we want to keep the probe effect of using the profiler as low as we can: we do not want to introduce or mask conflicts by enabling or disabling profiling.

We identify three main techniques for profiling TM applications. The first technique identifies multiple conflicts from a single program run and associates each conflict with contextual information. The contextual information is necessary to relate the wasted work to parts of the program as well as constructing the winner and victim relationship between the transactions. The second technique identifies the data structures involved in conflicts, and it associates the contended objects with the different places where conflicting accesses occur. The third technique visualizes the progress of transactions and summarizes which transactions conflict most. This is particularly useful when first trying to understand a transactional workload and to identify the bottlenecks that are present.

Our profiling framework is based on the Bartok-STM system [55]. Bartok is an ahead-of-time C# compiler which has language-level support for TM. Where possible, the implementation of our profiling techniques aims to combine work with the operation of the C# garbage collector (GC). This helps us reduce the probe effect because the GC already involves synchronization between program threads, and drastically affects the contents of the processors' caches; it therefore masks the additional work added by the profiler. Although we focus on Bartok-STM, we hope that the data collected during profiling is readily available in other TM systems.

This chapter continues in Section 5.1 with a discussion on the problems which motivated the study on the profiling techniques for transactional applications. Section 5.2 surveys the state of the art in the area and relates this work to the others. Section 5.3 introduces the profiling techniques. Section 5.4 describes the design and implementation of the profiling framework and evaluates its overhead and probe effect. Section 5.5 summarizes this chapter.

5.1 Motivation

The primary motivation for this work was the experience of building AtomicQuake. Despite the existing experiments which reported reasonable TM overhead, AtomicQuake had high single threaded overhead ranging between 4 and 5 times slowdown over the base non-TM version. It was very difficult and even possible to understand what are the exact reasons the high overhead. There were two reasons of making AtomicQuake difficult to profile. First, the STM runtime did not provide profiling at the sufficient level of detail to draw complete conclusions. Second, the environment we used (i.e. the prototype version of Intel C/C++ with STM support) were closed source and we did not have clear understanding about its low level operation.

The conclusion after encountering these these problems was that programmers need:

1. systematic way of obtaining profiling information from the underlying TM (i.e. profiling framework);

2. profiling results which are reported in a form independent from the underlying TM implementation; and

3. profiling results give comprehensive information about the TM related bottlenecks – i.e. where, when and why conflicts happen.

Later studies also supported these findings. Pankratius [92] and Rossbach [100] report that very first versions of TM programs perform poor because programmers tend to use large atomic blocks because they focus on the correct implementation of the synchronization and ignore the performance.

These problems are explored and addressed with the work described in this chapter.

5.2 Related Work

Chafi *et al.* developed the Transactional Application Profiling Environment (TAPE) which is a profiling framework for HTMs [23]. The raw results that

TAPE produces can be used as input for the profiling techniques that we have proposed. This would enable profiling transactional applications that execute on HTMs or HyTMs.

In a similar manner, the Rock processor provides a status register to understand why transactions abort [34] (reflecting conflicts between transactions, and aborts due to practical limits in the Rock TM system). Examples include transactions being aborted due to a buffer overflow or a cache line eviction. Profiling applications in this way is complementary to our work which will allow users to further optimize their code for certain TM system implementations.

Concurrent with our own work, Chakrabarti [25] introduced dynamic conflict graphs (DCG). A coarse grain DCG represents the abort relationship between the `atomic` blocks similar to aborts graph (see Figure 5.7). A fine grain DCG represents the conflict relationship between the conflicting memory references. To identify the conflicting memory references, Chakrabarti proposed a technique similar to basic conflict point discovery (which is described in this chapter). Our new extensions over basic conflict point discovery (Section 5.3.2) would generate more complete DCGs. The more detailed fine grain DCGs would complement the profiling information by linking the symptoms of lost performance to the reasons at finer statement granularity. In addition, identifying conflicting objects is another feature which relates the different program statements where conflicts happen with the same object and vice versa.

Independently from us, Lourenço *et al.* [76] have developed a tool for visualizing transactions similar to the transaction visualizer that we describe in Section 5.3.5. They also summarize the common transactional characteristics that are reported in the existing literature such as abort rate, read and write set, etc. over the whole program execution. Our work complements theirs by reporting results in source language such as variable names instead of machine addresses. Also, we provide local summary which is helpful for examining the performance of specific part of the program execution.

Sonmez *et al.* [111] have profiled Haskell-STM applications using per-`atomic` block statistics. We extend this work by providing mechanisms to obtain statistics at various granularity, including per-transaction, per-`atomic` block, local and

global summary. In addition, our statistics include contextual information comprising the function call stack which is displayed via the top-down and bottom-up views. The contextual information helps relating the conflicts to the many control flows in large applications where `atomic` blocks can be executed from various functions and where `atomic` blocks include library calls.

In this earlier work, we also explored the common statistical data used in the research literature to describe the transactional characteristics of the TM applications: time spent in transactions, read set, write set, abort rate, etc. In addition we generate a histogram about how much of the transactions' execution interleave. This information is particularly useful to see the amount of parallelism in the program and find cases when a program does not abort but also does not scale.

5.3 Profiling Techniques

As with any other application, factors such as compiler optimizations, the operating system, memory manager, cache size, etc. will effect on the performance of programs using TM. However in addition to these factors, performance of transactional applications also depends on 1) the performance of the TM system itself (e.g., the efficiency of the data structures that the TM uses for managing the transactions' read-sets and write-sets), and 2) the way in which the program is using transactions (e.g., whether or not there are frequent conflicts between concurrent transactions).

Figure 5.1 provides a contrived example to illustrate the difference between TM-implementation problems and program-specific problems. The code in the example executes transactional tasks (line 4) and, depending on the task's result, it updates elements of the array `x`. This code would execute slowly in TM systems using naïve implementations of lazy version management: every iteration of the `for` loop would require the TM system to search its write set for the current value of variable `taskResult` (lines 6 and 8). This would be an example of a TM-implementation problem (and, of course, many implementations exist that support lazy version management without naïve searching [56]). On the other hand, if the programmer had placed the `while` loop inside the `atomic` block, then

the program's abort rate would increase regardless of the TM implementation. This would be an example of a program-specific problem.

This research focuses on this second kind of problem. The rationale behind this is that reducing conflicts is useful no matter what kind of TM implementation is in use; optimizing the program for a specific TM implementation may give additional performance benefits on that system, but the program might no longer perform as well on other TM systems.

In this section we describe our profiling techniques for transactional memory applications. These profiling techniques operate on typical TM data and are not restricted to our profiling framework only. Therefore, the ideas described here are also applicable for other STMs and HTMs. We follow two main principles. First, we report the results at the source code language such as variable names instead of memory addresses or source lines instead of instruction addresses. Results presented in terms of structures in the source code are more meaningful as they convey semantic information relevant to the problem and the algorithm. Second, we want to reduce the probe effect introduced by profiling, and to present results that reflect the program characteristics and are independent from the underlying TM system. For this purpose, we exclude the operation time of the TM system (e.g. roll-back time) from the reported results.

5.3.1 Basic Conflict Point Discovery

Conflict point discovery is a technique to identify the statement where a transaction is detected to be invalid and consequently aborted. Results from conflict point discovery are reported in source code level (see Figure 5.2) and includes information how many times a given statement has been involved in a conflict.

We support conflict point discovery by using further "stub" functions to provide abstraction over the underlying STM library. These stubs are called when the STM library does book-keeping work. In effect, this automates the *reach point* technique we used in earlier work [42], by removing the need for manual instrumentation of code. We experimented with an alternative implementation that operates entirely on the debugger side, but the overhead of additional internal breakpoints was prohibitively high.

```
    int taskResult = 0;
.
 1: while (!taskQueue.IsEmpty) {
 2:    atomic {
 3:       Task task = taskQueue.Pop();
 4:       taskResult = task.Execute();
 5:       for (int i < 0; i < n; i++) {
 6:          if (x[i] < taskResult) {
 7:             x[i]++;
 8:          } else if (x[i] > taskResult) {
 9:             x[i]--;
10:          }
11:       }
12:    }
13: }
```

Figure 5.1: An example loop that atomically executes a task and updates array elements based on the task's result. The repeated use of `taskResult` value at lines 6 and 8 would expose the TM specific overheads between lazy versioning (i.e. buffered updates) and eager versioning (i.e. in-place updates) TMs. Lazy versioning TMs would execute this code fragment slower because the TM would need to obtain the most recent value of `taskResult` by searching in the write buffer. On the other side, this code would have a program specific bottleneck if the programmer had conservatively put the whole `while` loop inside the `atomic` block.

```
File:Line          #Conf.  Method     Line
-------------------------------------------------------------
Hashtable.cs:51    152     Add        if (_container[hashCode] ...
Hashtable.cs:48    62      Add        uint hashCode = HashSdbm ...
Hashtable.cs:53    5       Add        _container[hashCode] = n ...
Hashtable.cs:83    5       Add        while (entry != null)
ArrayList.cs:79    3       Contains   for (int i = 0; i < cont ...
ArrayList.cs:52    1       Add        if (count == capacity - 1)
```

Figure 5.2: Example output generated by conflict point discovery for the C# version of Genome application.

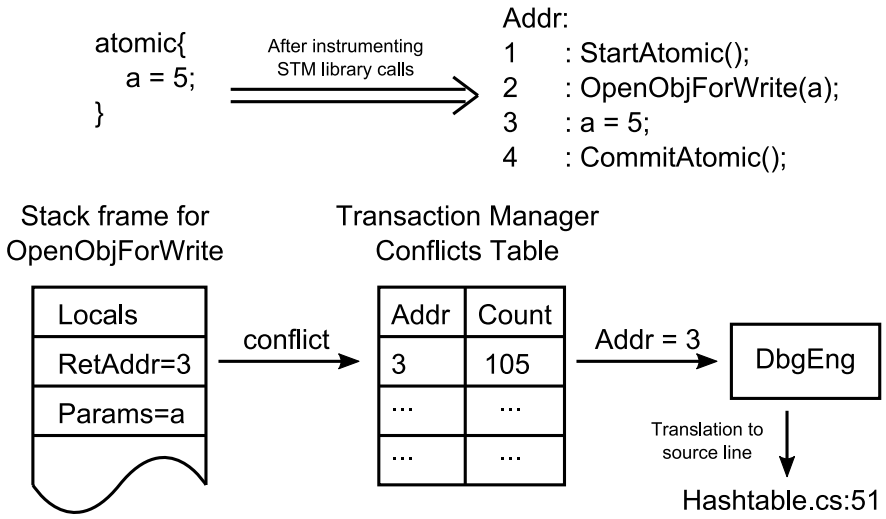


Figure 5.3: This figure shows how we identify the locations in the source code where conflicts happen. We modified the read and write barriers in the STM library to log their return address in the user code. When conflict is detected, we record the return address associated with the conflicting memory access in *Conflicts Table* and increment the conflict counter. At the end of the execution, using the debugger engine (DbgEng) we translate the addresses into source lines.

In Bartok-STM, conflicts can be detected in the write barriers, intermediate validations of the read set and the commit method (which also validates the read set). In the stubs we add to read and write barriers, we log the return address of the STM operation, along with the address of the memory location being accessed (see Figure 5.3). The return address of these functions is the place in the user code where the actual access to the memory is done. If the STM library detects a conflict while handling any of these methods we record the return address together with the origin of the conflict—whether caused by read or write. If the address is already recorded, then we increment a conflict counter associated with it.

Next section describes extensions over this *basic* conflict point discovery which assign additional context to help identify the hot control flow which causes most conflicts.

```
increment() {
   counter++;
}
```

```
probability80() {
   ...
   probability = random() % 100;
   if (probability < 80)
      atomic {
         increment();
      }
   ...
}
```

```
probability20() {
   ...
   probability = random() % 100;
   if (probability >= 80) {
      atomic {
         increment();
      }
   ...
}
```

```
.
Thread 1
.
for (int i < 0; i < 100; i++) {
    probability80();
    probability20();
}
```

```
Thread 2

for (int i < 0; i < 100; i++) {
    probability80();
    Probability20();
}
```

Figure 5.4: In this example code two threads call functions which increment a shared counter with different probability. Basic conflict point discovery will only report that all conflicts happen in function `increment`. However without knowing which function calls `increment` most the user cannot find and optimize the critical path. In this example the critical path would be `probability80` – `increment`.

5.3.2 Advanced Conflict Point Discovery

The previous section introduced a "conflict point discovery" technique that identifies the first program statements involved in a conflict. However, after using this technique to profile applications from STAMP, we identified two limitations: 1) it does not provide enough contextual information about the conflicts and 2) it accounts only for the first conflict that is found because one or other of the transactions involved is then rolled back.

In small applications and micro-benchmarks most of the execution occurs in one function, or even in just a few lines. For such applications, identifying the statements involved in conflicts would be sufficient to find and understand the TM bottlenecks. However, in larger applications with more complicated control flow, the lack of contextual information means that basic conflict point discovery would only highlight the symptoms of a performance problem without illuminating the underlying causes.

For example, in Figure 5.4 the two functions `probability80` and `probability20` atomically increment a shared counter by calling the function `increment` with a probability of 80% and 20%. When `probability80` and `probability20` are called in a loop by two different threads, basic conflict point discovery will report that all conflicts happen inside the function `increment`. But this information alone is not sufficient to reduce conflicts because the user would need to distinguish between the different stack back-traces that the conflicts are part of. In this case, the calls involving `probability80` should be identified as more problematic than those going through `probability20`. Similarly, for other transactional applications, the reasons for the poor performance would most likely be for using, for example, inefficient parallel algorithms, using unnecessarily large `atomic` blocks, or using inappropriate data structures which allow low degrees of concurrent usage.

The second disadvantage of basic conflict point discovery is that it only identifies the first conflict that a transaction encounters. It is possible that two transactions might conflict on a series of memory locations and so, if we account for only the first conflict, the profiling results will be incomplete. As a consequence, the user will not be able to properly optimize the application and most likely will

```
   // Thread 1                  // Thread 2
1: atomic {                     atomic {
2:    obj1.x = t1;                 ...
3:    obj2.x = t2;                 ...
4:    obj3.x = t3;                 ...
5:    ...                          obj1.x = t1;
6:    ...                          obj2.x = t2;
7:    ...                          obj3.x = t3;
8: }                            }
```

Figure 5.5: Basic conflict point discovery would only display the first statements where conflicts happen. On the given examples these statements are line 2 for Thread 1 and line 5 for Thread 2. However, the remaining statements are also conflicting and most likely revealed on the subsequent profiles.

need to repeat the profiling several times until all the omitted conflicts are revealed. The programmer can end up needing to "chase" a conflict down through their code, needing repeated profile-edit-compile steps. Figure 5.5 provides an example: basic conflict point discovery would only identify the conflicts on `obj1` (line 2 for Thread 1 and line 5 for Thread 2). However, the remaining statements are also conflicting and most likely will be revealed by subsequent profiles once the user has eliminated the initial conflicting statements.

We address the described limitations namely by providing contextual information about the conflicts and accounting for all conflicting memory accesses within aborted transactions.

The contextual information comprises the `atomic` block where the conflict happens and the call stack at the moment when the conflict happens. It is displayed via two views: top-down and bottom-up (Figure 5.6). In both cases, each node in the tree refers to a function in the source code. However, in the top-down view, a node's path to the root indicates the call-stack when the function was invoked, and a node's children indicate the other functions that it calls. The leaf nodes indicate the functions where conflicts happen. Consequently, a function called from multiple places will have multiple parent nodes. Conversely, in the bottom-up view, a root node indicates a function where a conflict happens and its children nodes indicate its caller functions. Consequently, a function called from multiple places will have multiple child nodes. Furthermore, to help the

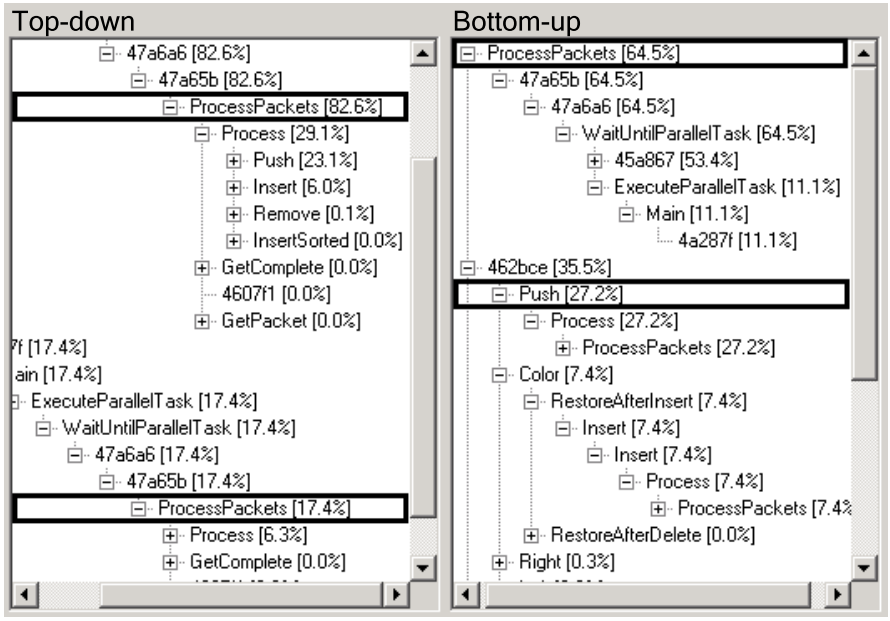


Figure 5.6: On the left is top-down tree view and on the right bottom-up tree view obtained from the 4-threaded execution of non-optimized Intruder application. The top-down view (left) shows that almost 100% (82.6%+17.4% summed from the two trees) of the total wasted work is accumulated at function `ProcessPackets`. The bottom-up view (right) shows that 64.5% of the total wasted work is attributed to function `ProcessPackets`, and 27.2% to function `Queue.Push` which is called from `ProcessPackets` and the rest to other functions. The non-translated addresses are internal library calls. Because of different execution paths that follow from the main program thread and the worker threads the top-down view draws 2 trees instead of 1.

programmer find the most time-consuming stack traces in the program, each node includes a count of the fraction of wasted work that the node (and its children) are responsible for.

To find all conflicting objects in an aborting transaction, we simply continue checking the remaining read set entries for conflicts. In the rare case, when the other transactions that are involved in a conflict are still running, we force them to abort and re-execute each transaction serially. This way we collect the complete read and write sets of the conflicting transactions. By intersecting the read and write sets, we obtain the potentially conflicting objects. Unlike basic conflict point discovery, our approach will report that all statements in the code fragment from Figure 5.5 are conflicts. Our profiling tool displays the relevant information about the conflicting statements and conflicting objects in the bottom-up view (Figure 5.6) and the per-object view respectively (Figure 5.8).

Besides identifying conflicting locations, it is important to determine which of them have the greatest impact on the program's performance. The next section introduces the performance metrics which we use to do this, along with how we compute them.

5.3.3 Quantifying the Importance of Aborts

The profiling results should draw the user's attention to the `atomic` blocks whose aborts cause the most significant performance impact. As in basic conflict point discovery, a naïve approach to quantify the effect of aborted transactions would only count how many times a given `atomic` block has aborted. In this case results will wrongly suggest that a small `atomic` block which only increments a shared counter and aborts 10 times is more important than a large `atomic` block which performs many complicated computations but aborts 9 times. To properly distinguish between such `atomic` blocks we have used different metric called *WastedWork*. WastedWork counts the time spent in speculative execution which is discarded on abort.

Besides quantifying the amount of lost performance, it is equally important that the profiling results surface the possible reasons for the aborts. For example, the Bayes application has 15 separate `atomic` blocks, one of which aborts

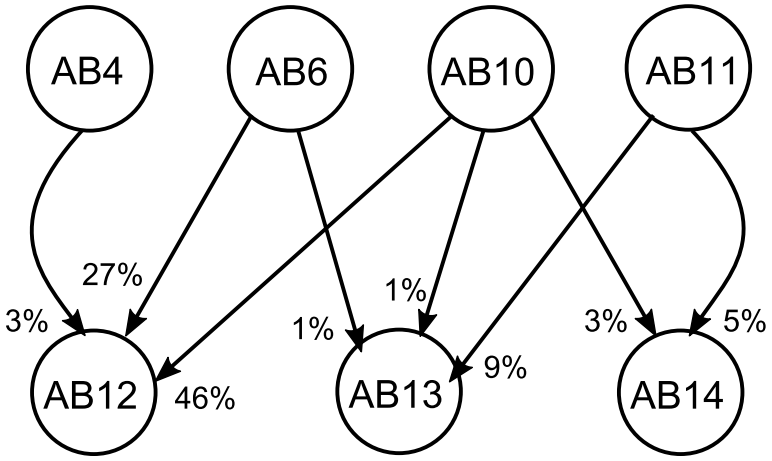


Figure 5.7: Example aborts graph from the execution of Bayes application. In this graph, 46% of the total wasted work in the program is when AB10 aborts AB12

much more frequently than the others (`FindBestInsertTask`). The WastedWork metric will tell us at which `atomic` block the performance is lost, but to reduce the number of aborts the user will also need to find the `atomic` blocks which cause `FindBestInsertTask` to abort. To mitigate this, we have introduced an additional metric *ConflictWin*. ConflictWin counts how many times a given transaction wins a conflict with respect to another transaction which aborts.

Using the information from the WastedWork and ConflictWin metrics, we construct the *aborts graph*; we depict this graphically in Figure 5.7, although our current tool presents the results as a matrix. The aborts graph summarizes the commit-abort relationship between pairs of `atomic` blocks; it is similar to Chakrabarti's dynamic conflict graphs [25] in helping linking the symptoms of lost performance to their likely causes.

5.3.4 Identifying Conflicting Data Structures

Atomic blocks abstract the complexity of developing multi-threaded applications. When using `atomic` blocks, the programmer needs to identify the atomicity in the program whereas using locks the programmer should identify the shared data structures and implement atomicity for the operations that manipulate them.

However, based on our experience using `atomic` blocks, it is difficult to achieve good performance without understanding the details of the data structures involved [42; 135].

If the programmer wants transactional applications to have good performance it is necessary to know the shared data structures and the operations applied to them. In this case the programmer can use `atomic` blocks in an optimal way by trying to keep their scope as small as possible. For example, as long as the program correctness is preserved, the programmer should use two smaller `atomic` blocks instead of one large `atomic` block or as in Figure 5.1 put the `atomic` block inside the `while` loop instead of outside. An existing work illustrated examples where smaller `atomic` blocks aborted less frequently and incurred less wasted work when they did abort [42; 67; 93].

In addition, the underlying TM system may support language-level primitives to tune performance, or provide an API that the programmer can use to give hints about the shared data structures. For example, Yoo *et al.* [130] used the `tm_waiver` keyword [89] to instruct the compiler to not instrument thread-private data structures with special calls to the STM library. In Haskell-STM [52] the user must explicitly identify which variables are transactional. To reduce the overhead of privatization safety, Spear *et al.* [114] have proposed that the programmer should explicitly tell which transactions privatize data [113]. We believe that profiling results can help programmers use these techniques by describing the shared data-structures used by transactions, and how conflicts occur when accessing them.

In small workloads which in total have few data structures, the results from conflict point discovery (Section 5.3.1) would be sufficient to identify the shared data structures. For example, in the STAMP applications, there are usually only a small number of distinct data structures, and it is immediately clear which transaction is accessing which data.

However, in larger applications, data structures can be more complex, and can also be created and destroyed dynamically. To handle this kind of workload, our prototype tool provides a tree view that displays the contended objects along with the places where they are the subject of conflicts (Figure 5.8). In the example,

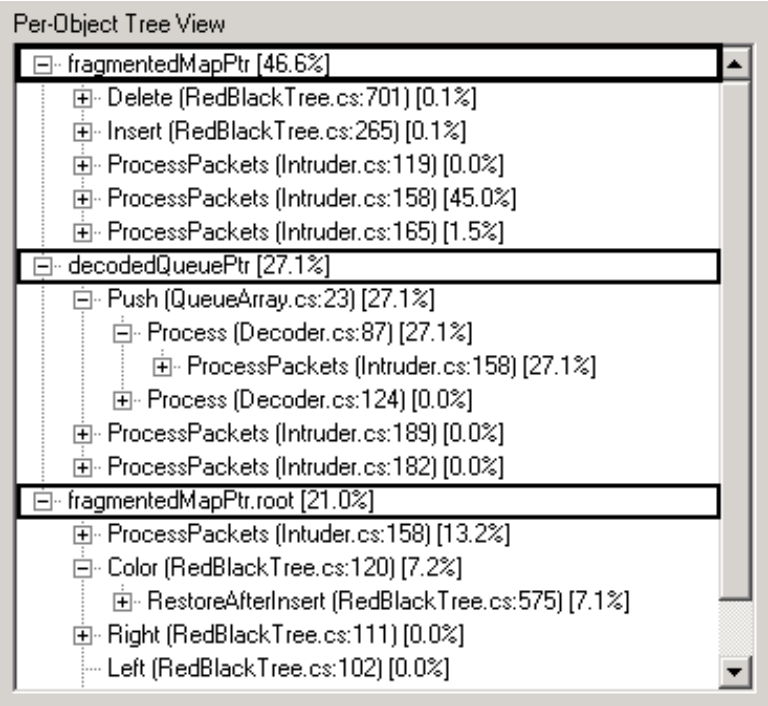


Figure 5.8: Per-object bottom-up abort tree. This view shows the contended objects and the different locations within the program where they have been involved in conflicts. Results shown are obtained from the 4 threaded execution of non-optimized Intruder application. For example, object `fragmentedMapPtr` has been involved in conflict at 5 different places - 3 in function `ProcessPackets`, 1 in `Delete` and 1 in `Insert`. Each object is also cumulatively assigned wasted work. Non-translated addresses are internal library calls.

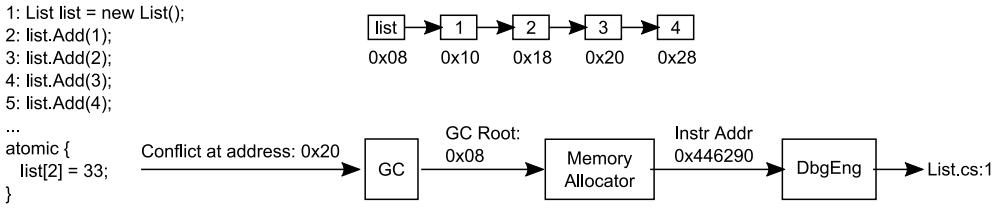


Figure 5.9: This figure demonstrates our method of identifying conflicting objects on the heap. The code fragment on the left creates a linked list with 4 elements. When the TM system detects a conflict in the `atomic` block, it logs the address of the contended object. During GC, the conflicting address is traced back to the GC root which is the list node. Then the memory allocator is queried at which instruction the memory at address "0x08" was allocated. At the end, by using the debugger engine the instruction is translated to a source line.

the object `fragmentedMapPtr` has been involved in conflicts at 5 different places which have also been called from different functions.

In our profiling framework we have developed an effective and low-overhead method for identifying the conflicting data structures, both static and dynamic. It is straightforward to identify static data structures such as global shared counters: it is sufficient to translate the memory address of the data structure back to a variable. However, it is more difficult when handling dynamically-allocated data structures such as an internal node of a linked list; the node's current address in memory is unlikely to be meaningful to the programmer.

For instance, suppose that the `atomic` block in Figure 5.9 conflicts while executing `list[2]=33` (assigning a new value to the third element in a linked list). To describe the resulting conflict to the programmer, we find a path of references to the internal list node from an address that is mapped to a symbol. This approach is similar to the way in which the garbage collector (GC) finds non-garbage objects. Indeed, in our environment, we map the conflicting objects to symbols by finding the GC roots that they are reachable from. If the GC root is a static object then we can immediately translate the address to a variable name. If the GC root is dynamically created, we use the memory allocator to find the instruction at which GC root was allocated and translate the instruction to a source line. To do this, we extended the memory allocator to record allocation

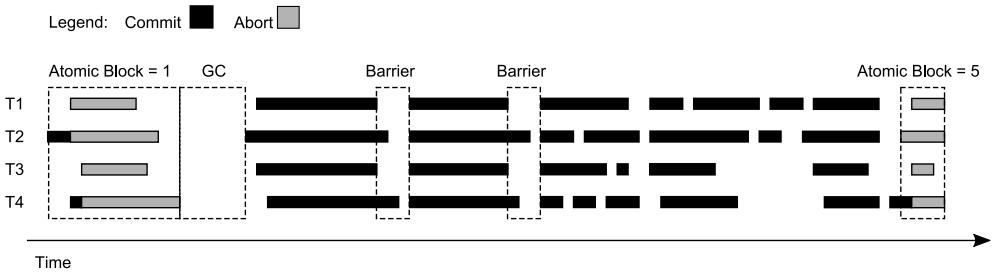


Figure 5.10: The transaction visualizer plots the execution of Genome with 4 threads. Successfully committed transactions are colored in black and aborted transactions are colored in gray. From this view, we can easily distinguish the different phases of the program execution such as regions with high aborts. By selecting different regions in this view, our tool summarize the profiling data only for the selected part of the execution. To increase the readability of the data, we have redrawn this figure based on a real execution.

locations.

5.3.5 Visualizing Transaction Execution

The next aspect of our profiling system is a tool that plots a time line of the execution of all the transactions by the different threads (Figure 5.10). In the view pane the transactions start from the left and progress to the right. Successfully committed transactions are colored black and aborted transactions are colored gray. The places where a color is missing means that no transaction has been running. The view in Figure 5.10 plots the execution of the Genome application from STAMP. From this view we can easily identify the phases where aborts are most frequent. In this case, most aborts occur during the first phase of the application when repeated gene segments are filtered by inserting them in a hashtable and during the last phase when building the gene sequence.

The transaction visualizer provides a high-level view of the performance. It is particularly useful at the first stage of the performance analysis when the user identifies the hypothetical bottlenecks and then analyzes each hypothesis thoroughly. Another important application of the transaction visualizer is to identify different phases of the program execution (e.g., regions with heavily aborting

transactions).

To obtain information at a finer or coarser granularity, the user can respectively zoom in or zoom out. Clicking at a particular point on the black or gray line displays relevant information about the specific transaction that is under the cursor. The information includes: read set size, write set size, `atomic` block id, and if the transaction is gray (i.e., aborted) it displays information about the abort. By selecting a specific region within the view pane, the tool automatically generates and displays summarized statistics only for the selected region.

Existing profilers for transactional applications operate at a fixed granularity [9; 23; 93; 111]. They either summarize the results for the whole execution of the program or display results for the individual execution of `atomic` blocks. Neither of these approaches can identify which part of a program's execution involves the greatest amount of wasted work. But looking at Figure 5.10 we can easily tell that in Genome transactions abort at the beginning and the end of the program execution.

The statistical information summarized for the complete program execution is too coarse and hides phased executions, whereas per-transaction information is too fine grain and misses conclusive information for the local performance. Obtaining local performance summary is important for optimizing transactional applications because we can focus on the bottlenecks on the critical path and then effectively apply Amdhal's law.

By using the transaction visualizer, the programmer can easily obtain a local performance summary for the profiled application by marking the region that (s)he is interested in. This will automatically generate summary information about the conflicts, transaction read and write set sizes, and other TM characteristics, but only for the selected region. The local performance summary from Figure 5.10 shows that aborts at the beginning of the program execution happen only in the first `atomic` block and aborts at the end of the program execution happen at the last `atomic` block in program order.

The global performance summary that our tool generates includes most of the statistics that are already used in the research literature. These are total and averaged results for transaction aborts, read and write set sizes, etc. In addition we build a histogram about the time two or more transactions were

executing concurrently. This histogram is particularly useful when diagnosing lack of concurrency in the program. For example, it is possible that a program has very low wasted work but it still does not scale because transactions do not execute concurrently.

5.4 Profiling Framework

We have implemented our profiling framework for the Bartok-STM system [55]. Bartok-STM updates memory locations in-place by logging the original value for rollback in case a conflict occurs. It detects conflicts at object granularity, eagerly for write operations and lazily for read operations. The data collected during profiling is typical of many other TM systems, of course.

The main design principle that we followed when building our profiling framework was to keep the probe effect and overheads as low as possible. We sample runtime data only when a transaction starts, commits or aborts. For every transaction we log the CPU timestamp counter and the read and write set sizes. For aborted transactions we also log the address of the conflicting objects, the instructions where these objects were accessed, the call stack of aborting thread and the `atomic` block id of the transactions that win the conflict. We process the sampled data offline or during garbage collection.

We have evaluated the probe effect and the overhead of our profiling framework on several applications from STAMP and WormBench (Table 5.1 and Table 5.2). To quantify the probe effect, we compared the application's overall abort rate when profiling is enabled versus the abort rate when profiling is disabled; a low probe effect is indicated by similar results in these two settings.

Our results suggest that profiling reduces the abort rate seen, but that it does not produce qualitative changes such as masking all aborts. These effects are likely to be due to the additional time spent collecting data reducing the fraction of a thread's execution during which it is vulnerable to conflicts. In addition, logging on abort has the effect of contention reduction because it prevents transactions from being restarted aggressively.

#Threads	Bayes+	Bayes-	Gen+	Gen-	Intrd+	Intrdr-	Labr+	Labr-	Vac+	Vac-	WB+	WB-
2	4.39	4.69	0.09	0.10	3.69	3.51	0.19	0.15	0.80	0.80	0.00	0.00
4	16.29	27.31	0.29	0.50	14.90	13.65	0.35	0.36	2.30	2.45	0.00	0.00
8	53.74	66.08	0.50	0.82	39.64	37.41	0.40	0.47	4.91	5.30	0.02	0.02

Table 5.1: The abort rate (in %) when the profiling is enabled ("+") and disabled ("-"). Results show that the profiling framework introduces small probe effect by reducing the abort rate for some applications. Results are average of 10 runs. Results for 1 are omitted because there are no conflicts.

#Threads	Bayes+	Bayes-	Gen+	Gen-	Intrd+	Intrdr-	Labr+	Labr-	Vac+	Vac-	WB+	WB-
1	1.59	1.00	1.28	1.00	1.29	1.00	1.07	1.00	1.26	1.00	0.71	1.00
2	1.00	0.56	0.92	0.65	0.97	0.58	0.64	0.61	0.83	0.59	0.60	0.55
4	0.23	0.23	0.91	0.50	0.91	0.36	0.45	0.46	0.58	0.40	0.41	0.33
8	0.21	0.20	0.72	0.50	1.57	0.38	0.72	0.56	0.53	0.34	0.33	0.22

Table 5.2: Normalized execution time with profiling enabled ("+") and profiling disabled ("-"). Results are average of 10 runs and normalized to the single threaded execution of the respective workload but with profiling disabled.

In applications with large numbers of short-running transactions, overheads can be higher as costs incurred on entry/exit to transactions are more significant. Profiling is based on thread-private data collection, and so the profiling framework is not a bottleneck for the applications' scalability.

5.5 Summary

This chapter introduced new techniques for profiling transactional applications. The goal of these profiling techniques is to help programmers find the bottlenecks specific to the program rather than the bottlenecks specific to the underlying TM system. To generate more comprehensive results we have extended our previous work on conflict point discovery. The extensions include metrics such as WastedWork and ConflictWin, assigning context to conflict points, building abort graphs, visualizing the transactions and identifying conflicting objects and data structures. We report all results in source code level such as variable names and statements.

Our profiling framework is based on Bartok-STM. The collected runtime data is common for the typical TM systems and can be obtained from other STMs and HTMs. Making the profiling framework less intrusive was one of our main design principles. Therefore we process the data offline or at runtime during garbage collection.

To examine the effectiveness of the proposed techniques we have profiled applications from STAMP TM benchmark suite and WormBench. Based on the profiling results we could successfully optimize Bayes, Labyrinth and Intruder. Bayes is an example where programs do not perform as expected when ported from non-object oriented environment such as C to object oriented environment such as C# or vice-versa. Labyrinth is an example where the programmer may give hints to the underlying TM system about the shared data structures and the operations applied on them. Intruder is an example of a program with poor performance which can be improved by using data structures with higher degree of parallelism and restructuring the code to reduce the wasted work.

Chapter 6

Optimizations

This chapter describe techniques to be used in a methodical approach for optimizing transactional applications. By applying these techniques the programmer can optimize the program to a specific TM implementation just like optimizing a program to a specific micro-architecture.

The optimization techniques are to be used after profiling a TM application and target performance improvements by reducing transaction abort rate and consequently wasted work. First, the programmer can try to change the location of the most conflicting write operations by moving them up or down within the scope of the `atomic` block. Depending on the underlying TM system, these changes may have significant impact on the overall performance making the application to scale well or bad (see Figure 6.14). Second, scheduling mutually conflicting `atomic` blocks to not execute in parallel would reduce the contention but when overused it may introduce new aborts and also serialize transactions. Third, checkpointing the transactions just before the most conflicting statements would reduce the wasted work by re-executing only the invalid part of the transaction. Forth, using pessimistic reads or treating transactional read operations as if they are writes can increase the forward progress in long running read-only transactions. Fifth, excluding memory references from conflict detection would increase the single-threaded performance and decrease aborts substantially. While the last approach might be very effective, applying it might be rather subtle because such transformations might not preserve the program correctness.

These optimization techniques can be automated through feedback directed compilation. Existing TM compilers [39; 52; 55; 89] and profilers [25; 76; 137] (see Section 5.4) can be extended to transparently pass compilation and profiling hints between each other. Depending on the transactional characteristics of the applications the compiler can adaptively apply different optimizations and choose the most suitable one after a certain number of iterations.

We describe how we ported a series of TM programs from C to C#. Initially, four of these applications did not scale well after porting (Bayes, Labyrinth and Intruder from the STAMP suite [20]). Profiling revealed that our version of Bayes had false conflicts due to Bartok-STM's object-level conflict detection. Another performance problem in Bayes was the wasted work caused by the aborts of the longest `atomic` block which is read-only. The remedy for the former problem was to modify the involved data structures and the remedy for the latter problem was to schedule the `atomic` block to not execute together with the `atomic` blocks which cause it to abort. Genome's performance suffered because of false conflicts on a congested hashtable. Its performance was brought to level by replacing a congested open addressing hashtable with a chaining hashtable. Labyrinth did not scale well because the compiler instrumented calls to the STM library for all memory accesses inside the program's `atomic` blocks. In contrast, the C version performed many of these memory accesses without using the STM library. We were able to achieve good scalability in the C# version by using *early release* to exclude the safe memory accesses from conflict detection. The authors of the STAMP benchmark suite report that Intruder scales well on HTM systems but does not scale well on some STMs. Indeed, initially, Intruder scaled badly on Bartok-STM. However, after replacing a contended red-black tree with a hashtable, and rearranging a series of operations, we achieved scalability comparable to that of HTM implementations. We also showed how to reduce wasted work by using nested `atomic` blocks. In Intruder, wrapping the most conflicting statements in nested `atomic` blocks reduces the wasted work from 45.5% to 36.8% (Table 6.5 versions Base and Nested Insert). Finally, we verified that our modified version of Intruder continued to scale well on other STMs and HTMs. These results illustrate how achieving scalability across the full range of current TM implementations can be extremely difficult. Aside from these example, the

remaining workloads we studied performed well and we found no further opportunities for reducing their conflict rates.

This chapter continues in Section 6.1 with the motivation for studying the optimization strategies and techniques for transactional application. Section 6.2 surveys the various technics used so far to optimize transactional applications. Section 6.3 introduces the profiling techniques. Section 6.4 discusses how the compiler can automatically compile the code with applying these optimization techniques. Section 6.5 demonstrates how we profiled and optimized transactional applications from the STAMP TM benchmark suite. Section 6.6 summarizes this chapter.

6.1 Motivation

Atomic blocks and transactions are yet knew programming abstraction and it is not studied how a programmer should tackle with the performance problems in programs which use them. Transactions add new types of bottlenecks to the applications which are specific to the TM programming model. Resolving these bottlenecks first require knowing where and why they happen and second knowing the underlying TM system. One type of the bottlenecks can be at higher application level. For example, using unnecessarily large `atomic` block instead of using two smaller `atomic` blocks or using data structure with lower degree of parallelism such as red black tree instead of a hashtable (e.g. for implementing a lookup table). Another type of bottlenecks can be at lower architectural level which depend on the implementation of the TM system. Example of such bottlenecks are false conflicts or how aborts are handled. The common between these two types of bottlenecks is the wasted work generated by aborting transactions. Once the nature of the TM bottlenecks in a program is known the programmer should follow a methodical approach for optimization which have the sole goal of reducing the wasted work. The work described in this chapter has the objective to demonstrate set of optimization techniques which can be used in a systematic way to reduce the wasted work in transactional applications.

6.2 Related Work

Adl-Tabatabai *et al.* [5] and Harris *et al.* [55] have described and implemented transactional memory optimizations in compilers with language level support of software transactional memory. Some of these leverage existing compiler optimizations such as loop transformations or common subexpression elimination on transactional code. Others are transactional memory specific and target detecting and eliminating redundant calls to the STM library such as repeated logging of the same object. For example when the compiler sees that an object is first read and then updated, then the compiler can skip instrumenting `OpenForRead` and instrument only one `OpenForWrite` call for both operations. This can be seen as being similar to using pessimistic reads (Section 6.3.4) however pessimistic reads can be used also for objects that are only read but not updated. Our optimization techniques are complementary and can be applied on a code which is already optimized by the compiler. Unlike automatic compiler optimizations, our techniques rely on prior profiling information about the program execution and the underlying TM implementation.

Bronson *et al.* [19] have used feedback directed compilation to optimize strongly isolated STM programs. In STMs, to provide strong isolation between transactional and non-transactional code, the non-transactional code should also be instrumented with calls to the underlying STM library. The compiler reads profiling data and instruments cheaper versions of STM calls for the non-transactional code that does not conflict but these calls are otherwise more expensive when a conflict happens. In Section 6.4 we describe how our optimization techniques (except early release) can be implemented in a such feedback directed compilation framework. While Bronson *et el.*'s optimizations are for the non-transactional code, ours are for the transactional code. Thus, they can be combined under the same framework to optimize both the transactional and non-transactional code.

To reduce aborts, Sonmez *et al.* [112] have interchangeably used pessimistic and optimistic reads in the Haskell runtime. Whenever an object becomes highly contended it uses pessimistic reads and whenever the object becomes less contended it switches back to optimistic reads. Identifying conflicting objects at runtime and switching between optimistic and pessimistic logging comes with

additional overhead. Using conflict point discovery, the programmer can easily identify the always conflicting objects and by using local transactional summaries the programmer can see when an object is contended and when not. In such cases the programmer can statically specify whether to open an object for read pessimistically and when to switch between pessimistic and optimistic reads. Static decisions can be used to exclude objects from dynamic decisions. This would reduce the runtime overhead of identifying conflicting objects and switching between two logging mechanisms for these objects. On the other side, dynamic decisions would increase the parallelism by switching between pessimistic and optimistic logging earlier than the static specification.

Several researchers have examined various methods for scheduling transactions dynamically [35; 37; 78; 129]. Typically transactions are continuously monitored for their abort frequency. Whenever the abort rate exceeds a certain threshold, transactions are serialized to reduce contention. Other approaches go one step further by keeping the history of the read and write sets of the transactions and try to predict weather two `atomic` blocks will conflict if they are executed concurrently. When possible the TM system may schedule two `atomic` blocks that are likely to conflict to execute on the same core. Unlike dynamic scheduling, static scheduling cannot be flexible and adapt to the changing behavior of transactions. However, static scheduling does not have runtime overheads and might perform better in cases when the transactional characteristics of `atomic` blocks are constant. In addition, these two approaches can be combined to complement each others' deficiencies – static scheduling can be used for the `atomic` blocks with predictive behavior and dynamic scheduling for those with non-predictive behavior.

Dice *et al.* [34] used privatization in a transactional implementation of a minimum spanning forest algorithm [66] to reduce the set of read set entries so that it entirely fits into the limited size hardware buffers of the Rock processor. In effect this optimization techniques reduces the probability of conflicts because it reduces the shared data on which transactions may operate.

Lupei *et al.* [77] have identified the set of memory which transactions access at runtime. Based on the analyzes they have dynamically scheduled transactions which operate on the same memory range to execute on the same thread. This

kind of optimization prevents two transactions which are potentially about to conflict to execute concurrently.

6.3 Optimization Techniques

In this section we describe several approaches to optimize transactional memory applications. These optimization techniques are TM implementation specific and changes on the code that favor one TM may have no effect or even perform worse on other TMs. Therefore, to properly apply these optimizations the programmer should be familiar with the implementation details of the underlying TM system. These optimization approaches are analogous to optimizing an application for a specific micro-architecture, for example, optimizing for the L1 cache size or the CPU's branch predictor.

6.3.1 Moving Statements

Moving statements such as hoisting loop invariants outside of a loop is a pervasive technique that optimizing compilers apply. Similarly, to reduce the cache miss rate, one can decide to pre-fetch data by manually moving a memory reference statement up in the code. Analogous to these examples, TM applications can also perform better by simply moving assignment statements (or statements that update memory) up or down in the code. Figure 6.14 plots the execution time of the Intruder application from the STAMP [20] benchmark suite using Bartok-STM [55]. In *Beginning* a call to a method which pushes an entry to a queue is moved to the beginning of the `atomic` block, and in *End* the call to the same method is moved to the end of the `atomic` block. Figure 6.1 is a contrived code example which represent how the code changes in *Beginning* and *End* look like.

The reason for the performance difference lies in the way how memory updates are handled by the TM system. In Bartok-STM, all update operations first lock the object and keep it locked until commit. If the requesting transaction sees that another transaction has already locked the object for update it aborts itself. In STMs like Bartok-STM and TinySTM [39] with encounter time locking, updates at the beginning of an `atomic` block on a highly contended shared variable such as

```
   // Beginning
1: atomic {
2:    counter++
3:    <statement 1>
4:    <statement 2>
5:    <statement 3>
6:    ...
7: }
8:
9:
       (a)
```

```
// End
atomic {
   <statement 1>
   <statement 2>
   <statement 3>
   ...
   counter++;
}
```

(b)

```
// Nested
atomic {
   <statement 1>
   <statement 2>
   <statement 3>
   ...
   atomic {
      counter++;
   }
}
```

(c)

Figure 6.1: A code where the increment of the shared counter is: (a) moved up (hoisted) to the beginning of the `atomic` block, (b) moved down to the end of `atomic` block, and (c) wrapped inside a nested `atomic` block.

a shared counter (Figure 6.1 (a)) may have the effect of a global lock. When one transaction successfully locks the object it will keep the lock until commit. In the mean time all the threads that try to execute the same `atomic` block will not be able to acquire the object's lock and will abort. This will serialize the program execution at this point. On the other hand, when the same update operation is at the end of the `atomic` block (see Figure 6.1 (b)) the transaction will keep the object locked for short time thus allowing other threads to execute the code concurrently until the problematic statement.

Because the approach of improving performance by moving the location of the statements relies on detecting WaW conflicts eagerly, it may not have effect on other TM systems. For example, when executed on the TL2 STM library [33], the location of the same statement affects the performance comparatively much less (see Figure 6.15). TL2 buffers updates and detects all types of conflicts lazily at commit time.

To identify exactly which statements to move, we used a profiling tool for TM applications built for Bartok-STM. This tool identifies the conflicting statements and assigns how much work is wasted at these statements in similar way as described in Chapter 5 and by Chakrabarti [25]. A statement which updates the memory and causes large wasted work would be a candidate for moving its location. However, the changes that the programmer makes should preserve the program correctness.

6.3.2 Atomic Block Scheduling

The purpose of transaction scheduling is to reduce the contention for performance. There is significant research on how transaction scheduling can be automated but to the best of our knowledge the problem of scheduling `atomic` blocks statically has not been studied.

Dynamic transaction scheduling introduces overhead at runtime because of the additional bookkeeping necessary to decide how to schedule the transactions. Static scheduling does not introduce such overheads. In addition, the scheduling requirements of a transactional application may be simple and not require any adaptive runtime algorithms. For example, Bayes from STAMP TM benchmark suite [20] has 15 `atomic` blocks but almost all the wasted work in the application is caused only by two `atomic` blocks that abort each other. For this case, a decision to statically schedule the two `atomic` blocks to not execute at the same time would be trivial. To decide exactly which `atomic` blocks to schedule, the programmer needs to know the `atomic` block which is responsible for the major part of the wasted work as well as the list of the other `atomic` blocks that it conflicts with. Such information can be obtained through conflict discovery graphs [25] or abort graphs from Section 5.3.3 (see Figure 6.7). However, the programmer should be aware that scheduling may not always deliver the expected performance. It is possible that after setting a specific schedule new conflicts appear or the program execution serializes.

6.3.3 Checkpoints

Various mechanisms have been proposed to implicitly checkpoint transactions at runtime [17; 122]. If a checkpointed transaction aborts, it is rolled back up to the earliest valid checkpoint. Checkpoints can improve the performance of transactional applications because (i) the transaction is not re-executed from the beginning and (ii) the valid checkpoints are not rolled back. The latter is particularly important for in-place update (i.e. eager versioning) TM systems because rollback operations are expensive. For example, suppose that we checkpoint the code in Figure 6.1 (b) at line 5. If conflict is detected at line 6 when incrementing

the `counter` and the remaining part of the transaction (i.e. lines 1–5) is valid, then only the increment will be rolled back and re-executed.

Techniques to automatically checkpoint transactions exists, but to the best of our knowledge there is no study on statically placing checkpoints. In the ideal case, transactions would re-execute only the code that is not valid. To achieve this, every transactional memory reference should be checkpointed, however this would cause excessive overhead. Therefore, it is necessary to identify where exactly to checkpoint a transaction. Good checkpoint locations are just before the memory references that cause most of the conflicts. We can easily identify these locations by using tools for profiling transactional memory applications such as conflict point discovery (Sectionr̃efsec:ch5:ProfilingTechniques). Conflict point discovery is a technique that identifies the statements that are involved in a conflict and quantifies their importance based on the wasted work. The programmer can manually checkpoint transactions just before the statements that cause most of the conflicts. Similarly to a transaction scheduling (Section 6.3.2), static checkpointing can be combined with dynamic checkpointing to off-load the runtime for the known conflicts.

Table 6.4 and Table 6.5 show the effect of checkpointing an `atomic` block in Intruder. In this experiment we used nested `atomic` blocks as shown in Figure 6.1 (c) because our STM library did not have checkpointing mechanisms. In this case, if the nested `atomic` block is invalid but the code in the outer block is valid, only the nested `atomic` block will re-execute. In effect this is the same as checkpointing at line 5 in Figure 6.1 (a).

As we can see, one can implement checkpoints by combining the use of `atomic` blocks. Furthermore, unlike checkpoints, nested `atomic` blocks are composable and can be used in functions that are called within other `atomic` blocks or outside `atomic` blocks [54].

6.3.4 Pessimistic Reads

To detect conflicts between transactions, the underlying TM implementation needs to know which memory references are accessed for read and for write. High performance STMs are not obstruction-free [38; 45], an implication of such

```
   // AB1                          // AB2
1: atomic {                       atomic {
2:    local_X = X;                   X++;
3:    <statement 1>               }
4:    ...
5:    <statement N>
6: }
```

Figure 6.2: AB1 is a long running `atomic` block which uses the value X and AB2 is a short running `atomic` block which increments X. If AB1 and AB2 execute concurrently, AB1 will be most of the time aborted by AB2.

design would allow one transaction be always aborted by another transaction. For example, consider a simple program of two `atomic` blocks AB1 and AB2. Suppose that AB1 is a long running transaction which uses the value of a shared variable X to perform complicated operations and AB2 has only a single instruction which increments X. In this case, AB2 will cause AB1 to abort repeatedly because AB1 will not be able to reach the commit point before AB2 (Figure 6.2).

To overcome this problem the user may use pessimistic reads or treat read operations as if they are writes. In the first approach it is necessary to update all transactional references to X with the proper pessimistic read operations. Without compiler support, finding all such references manually might be difficult and in some cases impossible. The latter approach is less intrusive because the programmer does not need to update the other references to X. Using pessimistic reads or opening X for write in AB1 from Figure 6.2 would subsequently cause AB2 to abort and let AB1 to make forward progress. However, this kind of modification, while providing forward progress for AB1, may cause the remaining `atomic` blocks to abort more than they did before and therefore not have any performance improvement. For example, Bartok-STM pessimistically locks objects on write. In this case, if AB1 locks X at the beginning of the `atomic` block, then all other executions of AB2 will abort trying to open X for write. But if AB1 executes less frequently than AB2, then AB1 winning the conflicts over AB2 would be better.

We can find conflicting read operations such as X in AB1 from Figure 6.2 by looking at the results of conflict point discovery. From these results we can

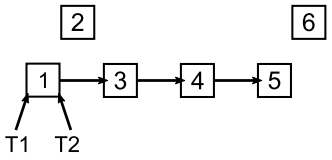
explicitly tell the compiler to open the read operations involved in many conflicts for write. However, the programmer should use such operations carefully because they may introduce new conflicts which might have negative performance impact.

6.3.5 Early Release

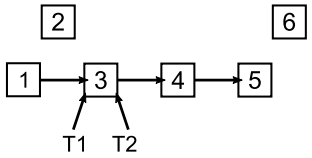
Early release is a mechanism to exclude entries in the transaction's read set from conflict detection [108; 110]. In certain applications it is possible that the final result of an `atomic` block is still correct although the read set is not valid. For example, consider an `atomic` block which inserts entries in a sorted linked list (Figure 6.3). Thread T1 wants to insert value 2 and thread T2 wants to insert value 6. To find the right place to insert the new values the two threads iterate over the the list nodes and consequently add them to the transaction's read set. T2 aborts because T1 finishes faster and after inserting the new node it invalidates T2's read set. However, T2 could still correctly insert the node although some entries in its read set are invalid. In this case we can exclude all nodes except 5 from conflict detection.

After carefully studying the Lee's path routing algorithm, Watson *et. al.* [125] have used early release to exclude a major part of the transaction's read set from conflict detection. To achieve similar results, Yoo *et al.* [130] instructed the compiler and Cao Minh *et al.* [20] deliberately skipped inserting calls to the STM library while copying the shared matrix into a thread local variable in Labyrinth. Caching the values of shared variables to a thread local storage, as in Bayes, is another form of excluding the shared variables from conflict detection.

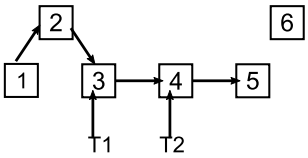
The experience of these studies reports that early release improves the application performance significantly. However, the programmer should not forget that it is not a safe operation (i.e. it can break program correctness). Applying this technique requires prior knowledge about the shard data structures used in the algorithm and the operations applied on them – namely whether or not the algorithm can be relaxed. The available profiling tools can help in identifying the shared objects that are involved in conflicts. Provided with this information, the programmer can focus on the specific objects and try to use early release when possible or use different implementations for the data structures.



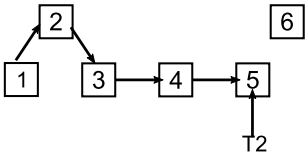
	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1		1	
T2	1		1	



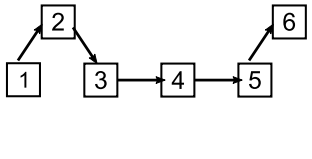
	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1, 3	1	1, 3	1
T2	1, 3		1, 3	



	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1, 3 COMMIT	1	1, 3 COMMIT	1
T2	1, 3, 4		1, 3, 4	



	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1, 3 COMMIT	1	1, 3 COMMIT	1
T2	1, 3, 4	5	1, 3, 4	5



	Without Early Release		With Early Release	
	Read Set	Write Set	Read Set	Write Set
T1	1, 3 COMMIT	1	1, 3 COMMIT	1
T2	1, 3, 4 ABORT	5	1, 3, 4 COMMIT	5

Figure 6.3: Transaction T1 inserts number 2 and transaction T2 inserts number 6 in sorted linked list. Without using early release T2 will abort and when using early release T2 will commit successfully.

```
   // Non-instrumented
1: atomic {
2:    local_x = shared_x;
3:    shared_y = 5;
4: }
5:
6:
      (a)
```

```
// Instrumented code
atomic {
   OpenForRead(shared_x);
   local_x = shared_x;
   OpenForWrite(shared_y);
   shared_y = 5;
}
     (b)
```

Figure 6.4: To track the memory references for later conflict detection, the compiler instruments calls to the STM library for the memory reads and updates. (a) non-instrumented `atomic` block, (b) `atomic` block instrumented with calls to OpenForRead and OpenForWrite respectively for the read and write memory references. The example is for in-place update STM.

6.4 Feedback Directed Compilation

Feedback directed compilation is a method of compiling a source code to a binary based on profiling data which is collected from previous program executions. This compilation approach has the advantage of identifying and optimizing the code segments that constitute a large part of the program execution thus improving the performance.

In this section we discuss how to apply the techniques from Section 6.3 for a feedback directed compilation. There already exist several compilers [39; 55; 89] and profilers [23; 76; 137] for transactional applications. Building a framework for transparent integration between the compiler and the profiler is a well understood engineering task which was studied by other researcher work [26; 69].

6.4.1 Moving Statements

In some cases, moving an assignment statement within the same `atomic` block can be automated. Moving the statement up in the code could be trivial. It would be sufficient that the compiler instruments the respective `OpenForWrite` operation earlier in the code and leaves the assignment operation in its place. In the example from Figure 6.4 (b), that would imply to instrument the call to `OpenForWrite` at line 2. However, the same approach cannot be used for moving a statement

down in the code order; calling `OpenForWrite` after a memory updated will not be correct. Moving statements down in the code would require additional code analysis to ensure that the program correctness is preserved. However, if the user manually moves down the problematic statement, the compiler can repeatedly move the place of `OpenForWrite` operation up and at the end choose the one which delivers the best performance.

Furthermore, similar automatic code reordering can be effectively applied to statements within functions called inside the `atomic` block by moving them within the function scope. Moving function calls within the scope of an `atomic` block would require inter procedural analysis which in some cases may not be sufficient.

6.4.2 Atomic Block Scheduling

Unlike moving statements around, the program correctness cannot be broken by scheduling the execution of `atomic` blocks. In this case, the task of the compiler would be simpler. The compiler can schedule the `atomic` blocks based on the profiling information obtained from aborts graph (Figure 6.7). The compiler can repeat various scheduling policies until obtaining the best performance [91; 120].

6.4.3 Checkpoints

Just like `atomic` block scheduling, checkpoints are also safe operations. The compiler can arbitrarily instrument checkpoint operations or wrap code segments within nested `atomic` blocks. A more advanced approach could be using abstract nested transactions (ANT) [53]. The process of automatic checkpoint instrumentation can be trivial as described in Section 6.3.3: the compiler will automatically instrument checkpoints just before the statements which are involved in more than a certain threshold number of conflicts.

6.4.4 Pessimistic Reads

Using pessimistic reads or opening read memory references for write is also a safe operation and can be easily performed by the available STM compilers. However, overuse of these operations may have negative impact on the performance due

to introducing new conflicts or serializing transactions [112]. The compiler can choose the statements based on the number of conflicts they have been involved in and the wasted time that these conflicts have caused. This information can be obtained from the fine grain conflict discovery graphs [25] or a conflict point discovery (Section 5.3). The compiler can selectively open the most conflicting read memory references for write. Because opening a specific object for write may create other conflicts, the compiler can combine several profiling histories when choosing the statements to open for write [103].

Currently, there are compiler optimizations that directly open a memory reference for write when there are cases of write after read [55]. This also saves the runtime from logging the same memory operation into both read and write sets.

6.4.5 Early Release

Early release is not a safe operation. To benefit from early release, deep knowledge about the problem and the solution is required. As far as we know there are no mechanisms to automate early release in transactional applications.

6.5 Case Studies

In this section we present a series of case studies of profiling and optimizing the performance of applications from the STAMP TM benchmark suite [20] and from the synthetic WormBench workload [134] by using our techniques. The goal of these case studies is to evaluate the effectiveness of our profiling and optimization techniques: namely wether the profiling techniques reveal the symptoms and causes of the performance lost due to conflicts in these applications and wether our optimization techniques indeed improve the performance of these applications.

To see whether our profiling and optimization techniques can be equally applied across a range of TM implementations we utilize two different STMs – TL2 [33] and Bartok-STM [55]. TL2 buffers speculative updates and detects conflicts lazily at commit time for both reads and writes. It operates at word granularity by hashing a memory address to transactional word descriptor. Bartok is an ahead of time C# to x86 compiler with language level support for STM.

#Threads	BayesNonOpt	BayesOpt	IntrdNonOpt	IntrdOpt	LabrNonOpt	LabrOpt
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.32	0.56	1.16	0.58	5.25	0.61
4	1.49	0.23	2.92	0.36	30.42	0.46
8	4.81	0.20	n/a	0.38	n/a	0.56

Table 6.1: The normalized execution time of Bayes, Labyrinth and Intruder before and after optimization. Results are average of 10 runs and the execution time for each applications is normalized to its single threaded execution time. "n/a" means that the application run longer than 10 minutes and was forced termination.

Bartok-STM updates memory locations in-place by logging the original value for rollback in case a conflict occurs. It detects conflicts at object granularity, eagerly for write operations and lazily for read operations.

For this experiment we have ported several applications from the STAMP suite from C to C#. We did this in a direct manner by annotating the `atomic` blocks using the available language construct that the Bartok compiler supports. In the original STAMP applications, the memory accesses inside `atomic` blocks are made through explicit calls to the STM library, whereas in C# the calls to the STM library are automatically generated by the compiler. WormBench is implemented in the C# programming language.

6.5.1 Bayes

Bayes implements an algorithm for learning the structure of Bayesian networks from observed data. Initially our C# version of this application scaled poorly (see Table 6.1). By examining the data structures involved in conflicts, we found that the most heavily contended object is the one used to wrap function arguments in a single object of type `FindBestTaskArg` (Figure 6.5(a)). Bartok-STM detects conflicts at object granularity, and so concurrent accesses to the different fields of the same object result in false conflicts. The false conflicts caused 98% of the total wasted work. With 2 threads the wasted work constituted about 24% of the program's execution, and with 4 threads it increased to 80%. We optimized the code by removing the wrapper object `FindBestTaskArg` and passing the function arguments directly (see Figure 6.5(b)). After this small optimization Bayes scaled as expected (Table 6.1).

From this point we wanted to see wether we can improve the performance of Bayes more. We noticed that out of 15 `atomic` blocks only one, `atomic` block AB12, aborts most and causes 93% of the total wasted work. AB12 calls the method `FindBestInsertTask` and from the per-`atomic` block statistics we could see that it is the longest read-only transaction. Aborts graph in Figure 6.6 shows that `atomic` block AB12 is always being aborted by a non-read-only `atomic` blocks AB6 and AB11. Most of the aborts of AB12 are caused by AB11. AB11 is a very short running `atomic` block which updates and caches the shared variables `baseLogLikelihood` and `numTotalParent` into a thread local variable. Based on this profiling information we have decided to statically schedule `atomic` blocks AB11 and AB12 to not execute in parallel. The results in Figure 6.7 showed to be slightly better but not encouraging because new pairs of aborting `atomic` blocks appeared. Now the aborts dominated between B10 and AB12 constituting 46% of the total wasted work. Despite adding an additional schedule between AB10 and AB12 the execution time did not get better while wasted work was evenly distributed among the non-scheduled `atomic` blocks.

Figure 6.8 is a histogram which shows the time when the execution of two or more transactions are overlapping and Figure 6.9 is a histogram which shows the number of active transactions at the moment when a new transaction starts. In the both figures we can see that scheduling `atomic` blocks limits the parallelism – fewer transactions overlap during execution (Figure 6.8) and there are fewer active transactions at the moment when a new transaction starts (Figure 6.9). Furthermore, in Figure 6.8 we can see that in the *Base* version (i.e. with no scheduling) about 35% of the time there is only one transaction executing and 14% of the time there are eight transactions executing in parallel. Considering that 83% of execution in Bayes is spent in transactions [20] the results from the histogram might suggest that the execution of transactions simply do not overlap. However, the actual reason is different. Bayes has few very long running `atomic` blocks and the remaining `atomic` blocks are comparably shorter (e.g. 100x to 10 000x shorter). Most of the time only one thread is executing one of these long transactions and the remaining threads execute the short transactions. This can be confirmed with the results from Figure 6.9. In the *Base* version 80% of the time when a new transaction stars there are already 7 other transactions running.

```
//Function declaration with wrapper object
Task FindBestInsertTask(FindBestTaskArg argPtr) {
   Learner learnerPtr            = argPtr.learnerPtr;
   Query[] queries               = argPtr.queries;
   ...
}
.
// Preparing a wrapper object
FindBestTaskArg argPtr = new FindBestTaskArg();
argPtr.learnerPtr      = learnerPtr;
argPtr.queries         = queries;
.
// Pass arguments with a wrapper object
FindBestInsertTask(argPtr);
                     (a)
.
// Function declaration with explicit parameters
Task FindBestInsertTask(
       Learner learnerPtr, Query[] queries, ...)
.
// Passing arguments without a wrapper object
FindBestInsertTask(learnerPtr, queries, ...)
                     (b)
```

Figure 6.5: Code fragments from Bayes: a) the original code with the wrapper object `FindBestTaskArg`; b) the optimized code with the removed wrapper object and passing the function parameters directly.

After we schedule AB11 (i.e. a short transaction) and AB12 (i.e. a 40 000 times longer transaction) to not execute in parallel the number of active transactions drops significantly.

6.5.2 Genome

In this section we describe how we iteratively optimized a C# version of the Genome application from the STAMP TM application suite [20]. We use conflict point discovery to examine how transactions progress.

Genome is a gene sequencing application implemented in C using TL2 STM library [33]. We initially ported this application from C to C# in a direct manner by annotating the `atomic` blocks using the available language constructs that the Bartok compiler implements. In the original version of Genome, the memory accesses inside `atomic` blocks are made through explicit calls to the STM

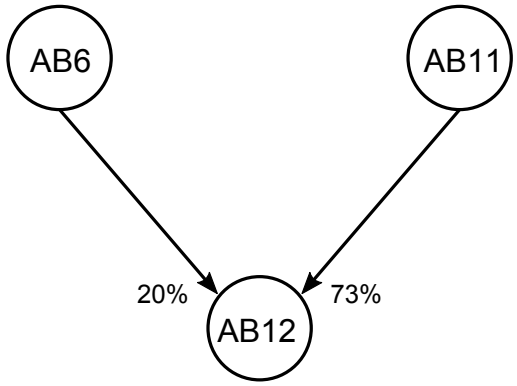


Figure 6.6: Aborts graph of Bayes before any schedule. In this graph we can see that AB12 is aborted by AB6 and AB12. Aborts between AB11 and AB12 cause 73% of wasted work in the program and aborts between AB6 and AB12 20% of the wasted work.

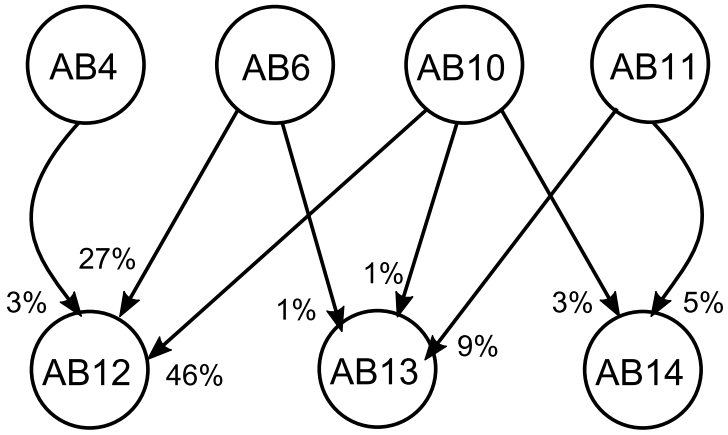


Figure 6.7: Aborts graph of Bayes when `atomic` blocks AB11 and AB12 are scheduled to not execute in parallel. In this figure AB10 aborts AB12 and the wasted work due to these aborts is 46% from the total program execution. Results are obtained from an execution with 8 threads.

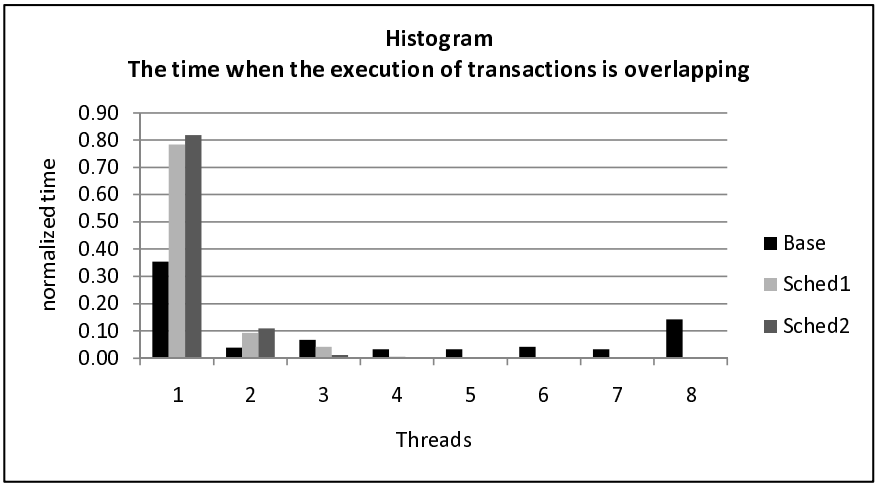


Figure 6.8: Bayes - this figure shows a histogram of the time when the execution of two or more transactions have overlapped.

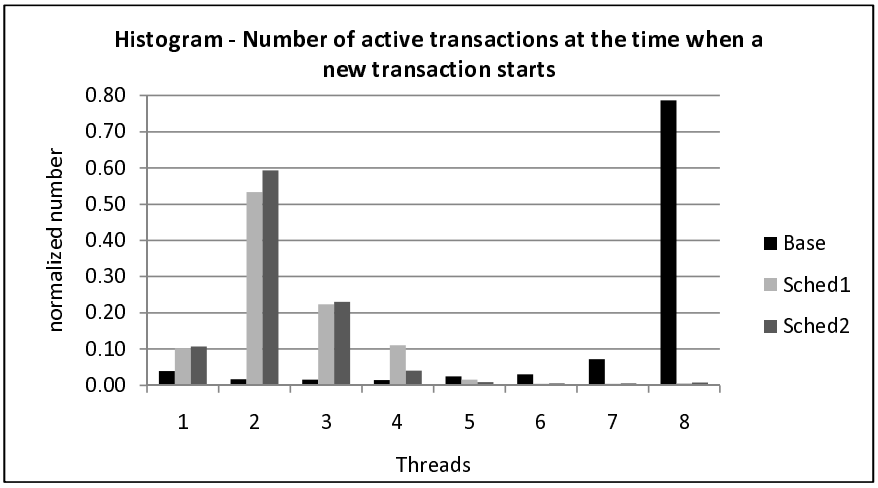


Figure 6.9: Bayes - this figure shows a histogram of the number of active transactions when a new transaction starts execution.

library, whereas in the C# port the STM library calls are automatically generated by the compiler. Our observations optimizing the C# version therefore do not necessarily reflect aspects of the manually-instrumented C program.

We performed our experiments on a 4*2-core CPU with 2 hardware threads per core. We show the effect of the different improvements on the normalized performance and on the reduction in the abort rate in Figure 6.10 and Figure 6.11 respectively. For comparison, we also show variants where we used a global lock in place of the `atomic` blocks (prefixed with L). We developed four variants using `atomic` blocks:

Unoptimized Genome (Unopt). Our first version of the C# Genome application had poor performance and did not scale. The reason for this was a very high abort rate. Using conflict point discovery, we saw that most of the conflicts happened in the first phase of the Genome application when duplicate gene segments are filtered by adding them to a hashtable. The highest contention was in two conflict points: 1) the test in a loop that checks whether a bucket already contains the entry to be added, and 2) when incrementing a shared counter that indicates the number of elements in the hashtable. After a careful look at the implementation of our hashtable we realized that it is a variation of an open addressing hashtable where entries are stored in the bucket array and the array is probed for empty slots on collisions.

Using chaining hashtable (Opt). The open addressing hashtable performs poorly in our implementation because Bartok-STM uses object level conflict detection: all array elements are considered as one object with respect to the conflict detection. We changed the implementation of the hashtable to a chaining version and also removed the shared counter, much like the hashtable from the STAMP suite. After these changes Genome's conflict rate was very low and scaled as in the original C version (see Figure 6.10 Opt).

Friendly fire pathology when rehashing. A second observation was that, when running with 4 or more threads, sometimes the execution was unusually long. Then looking at the number of re-executions of the individual `atomic`

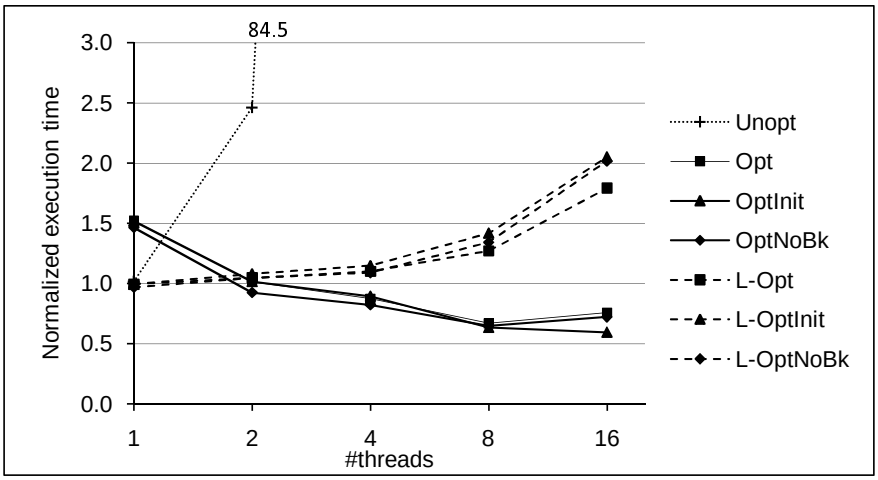


Figure 6.10: The execution time of Genome, normalized to L-Opt.

blocks, we observed the *friendly fire* pathology [18]: transactions were aborting one another without any being able to commit. Linking this information with the conflict points we found the underlying reason: one transaction was trying to rehash and at the same time another thread was starting the execution of the same `atomic` block. Then the two transactions were continuously aborting each other. When running with 2 threads it is less likely that the execution of the same `atomic` block will overlap, but with 4 or more threads this probability becomes much higher. Although a better solution could be found, our quick approach was to initialize the hashtable with a larger bucket array.

Initializing the buckets (OptInit). At this point, we examined the conflict data of the application more carefully and noticed that the number of conflicts when adding an element to a hashtable was approximately the same as the number of entries in the hashtable. Almost every addition of a new entry to the hashtable was causing a conflict. The reason for this was that we were initializing the elements in the bucket array at the time of adding the first entry in the bucket and again due to the object granularity conflict detection this was causing other transaction working on the array to abort. Our solution for this problem was to initialize the bucket array with default bucket objects during the initialization phase. This significantly reduced the abort rate (see Figure 6.11 OptInit) and

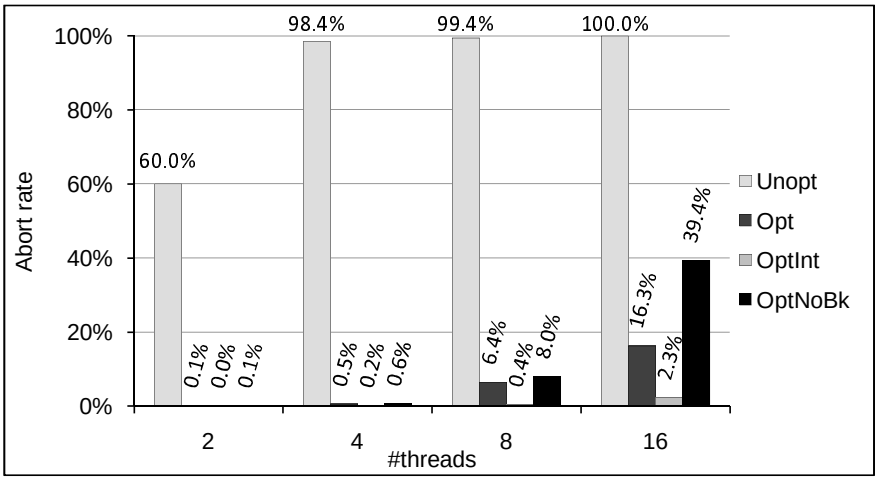


Figure 6.11: The effect of the optimizations on the abort rate.

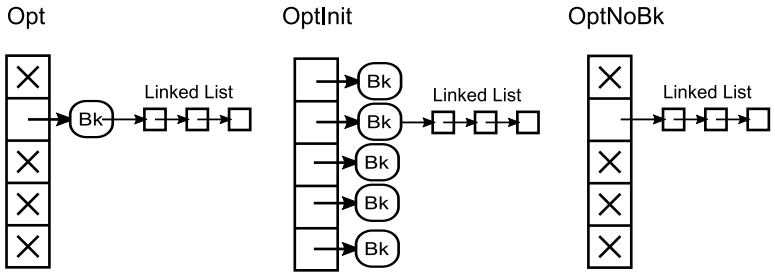


Figure 6.12: The different variants of the chaining hashtable we used in Genome. *Opt* uses bucket objects and does not initialize the bucket array. *OptInt* is the same as Opt but the bucket array is initialized. *OptNoBk* is a version of Opt that stores linked lists directly on the bucket array.

#Threads	TCC-Orig	TCC-Opt	Eazy-Orig	Eazy-Opt	TL2-Orig	TL2-Opt
1	1.00	1.01	1.00	0.96	1.00	0.80
2	0.73	0.67	0.61	0.59	0.92	0.60
4	0.51	0.43	0.37	0.35	0.63	0.48
8	0.39	0.31	0.26	0.22	0.65	0.52

Table 6.2: Execution time of Intruder before and after optimization on Scalable-TCC, Eazy-HTM and TL2. Results are average of 10 runs and normalized to the single threaded original version of Intruder.

made the application scale up to 16 threads.

Removing the buckets (OptNoBk). We have also developed a slightly different version of the chaining hashtable which does not have buckets and stores the linked list directly into the buckets array. This approach is slightly faster because it saves one indirection when performing a hashtable operation but has the same even higher contention than Opt. Figure 6.12 visualizes the implementation differences between the chaining hashtables that we used to optimize Genome.

We can see from Figure 6.11 that OptInit has smallest abort rate and scales up to 16 threads whereas Opt and OptNoBk scale up to 8 threads and are saturated at 16 threads. OptNoBk is faster because of saving one extra indirection due to the direct pointer in the array and not initializing all the buckets. In Figure 6.10 we can see that the single threaded execution of Unopt has the best performance but simply the implementation of this hashtable is not TM friendly.

6.5.3 Intruder

Intruder implements a network intrusion detection algorithm that scans network packets and matches them against a dictionary of known signatures. The authors of STAMP report that this application scales well on HTM systems but does not scale well on STMs [20]. Therefore understanding and eliminating the bottlenecks of this application was a challenge for us.

Our profiling techniques showed that the most contended objects in Intruder are `fragmentedMapPtr` and `decodedQueuePtr`. In 4-threaded execution, aborts in which `fragmentedMapPtr` was involved caused 67.6% wasted work and aborts in which `decodedQueuePtr` was involved caused 27.1% of wasted work. The

#Threads	AB1	AB2	AB3
1	0.00%	0.00%	0.00%
2	5.48%	91.01%	4.51%
4	3.38%	94.90%	1.72%
8	5.45%	93.43%	1.12%

Table 6.3: The wasted work caused by the aborts of the different `atomic` blocks in Intruder. Results are normalized.

wasted work of the both objects constituted 92.7% of the total program execution. The `fragmentedMapPtr` object is a map data structure used to reassemble the fragmented packets. Its implementation is based on red black tree and most important conflicts were happening during lookup. On the other hand, the lookup was invoked while adding a new entry to check if it already exists. Our approach of resolving the bottleneck at `fragmentedMapPtr` was to replace the underlying implementation with a chained hashtable. Unlike red black tree, when using hashtable transactions access fewer objects (i.e. their read set is smaller) and consequently have lower probability of conflict. We have experimentally verified that using hashtable instead of red black tree improves the application performance across different STM and HTM implementations (see Table 6.2). For this experiment we used state-of-the-art HTM systems (Scalable-TCC [24] and Eazy-HTM [119]) in a simulated environment.

Although we achieved satisfiable scalability for Intruder we continued to examine its performance in more depth. Intruder has in total three `atomic` blocks and our per-`atomic` block profiling showed that only one of them causes significant wasted work (Table 6.3). The subject `atomic` block contains only a call to method `Decoder.Process` (see Figure 6.13). We used our profiling tool to see exactly which statements from this `atomic` block are involved in conflicts. The results of conflict point discovery are shown in Table 6.4 (version Base).

Most of the conflicts in our system are read-after-write (RaW) or write-after-read (WaR) type and therefore detected at commit time (line 39). When the number of threads is low, significant amount of wasted work is caused due to conflicts at the statement which calls method `decodedQueuePtr.Push` (line 31). `decodedQueuePtr` data structure maintains the list of the packets which are assembled from several segments. Conflicts at this statement are of write-after-write

```
 1: public Error Process(Packet packetPtr) {
 2: ...
 3:    if (numFragment > 1) {
 4:       ...
 5:       if (fragmentedListPtr == null) {
 6:          ...
 7:       } else {
 8:          ...
 9:          fragmentedListPtr.InsertSorted(packetPtr);
10:          if (fragmentedListPtr.GetSize() == numFragment) {
11:             int i, numByte = 0;
12:             foreach (Packet fragmentPtr in fragmentedListPtr) {
13:                if (fragmentPtr.FragmentId != i) {
14:                   fragmentedMapPtr.Remove(flowId);
15:                   return Error.ERROR_INCOMPLETE;
16:                }
17:                numByte += fragmentPtr.Length;
18:                i++;
19:             }
20:
21:             char[] data = new char[numByte];
22:             int dst = 0;
23:             foreach (Packet fragmentPtr in fragmentedListPtr){
24:                Array.Copy(fragmentPtr.Data, data, dst);
25:                dst += fragmentPtr.Length;
26:             }
27:             Decoded decodedPtr = new Decoded();
28:             decodedPtr.flowId = flowId;
29:             decodedPtr.data = data;
30:
31:             decodedQueuePtr.Push(decodedPtr);
32:             fragmentedMapPtr.Remove(flowId);
33:          }
34:       }
35:    } else {
36:       ...
37:    } // end of if (numFragment > 1)
38:    return Error.ERROR_NONE;
39: }
```

Figure 6.13: Code fragment from Intruder. Method `Decoder.Process` is called inside an `atomic` block. Because of space constraints some irrelevant code such as initializations are omitted.

(WaW) type which Bartok-STM detects eagerly. When the number of threads increases, the wasted work at the call to method `fragmentedListPtr.InsertSorted` becomes dominant. `fragmentedListPtr` is a helper data structure (sorted list) used to assemble a packet from several segments. Conflicts at the call to `InsertSorted` are also WaW. Contention at this point increases with the number of threads because the probability of multiple threads inserting different segments belonging to the same packet increases.

We tried to reduce wasted work by moving the call to `Push` from the end of the `atomic` block (line 31) to the beginning of the `atomic` block (line 8). We anticipated that detecting conflicts earlier and aborting transactions earlier would generate less wasted work – speculative execution and state to rollback. However, opposite to our expectations the performance of the application degraded (see Figure 6.14). The conflict point analysis for the modified version showed that the poor performance is due to the increase in the number of re-executions and the abort rate of the `atomic` block (Table 6.4 version Push Move Up).

The reason for the increase in the number of re-executions and consequently the abort rate is specific to the implementation of Bartok-STM. When threads are about to update the `decodedQueuePtr` object, the TM system first locks the object. In this case when one thread successfully acquires object's lock all the other threads fail and abort until the lock is released during commit. In fact, the updates on `decodedQueuePtr` have the same effect as if it is a global lock. When the update is at the end of the `atomic` block (line 31) threads can execute large part of the `atomic` block concurrently, but when it is at the beginning of the `atomic` block (line 8) threads serialize trying to acquire the lock for `decodedQueuePtr`. The serialized execution is also confirmed by reading the histogram of the time when transactions are executed concurrently. However, on TM systems that detect WaW conflicts lazily at commit time such code changes do not have significant effect. We have performed the same experiment using TL2. In this case the performance of Intruder is similar in both cases (see Figure 6.15).

As discussed in Section 6.3.3, the high abort rate at the statements which call `Push` and `InsertSorted` suggests that using checkpoints or nested `atomic` blocks would improve the performance. We have carried three different experiments: 1) we have wrapped the call to `Push` in a nested `atomic` block (Table 6.4 version

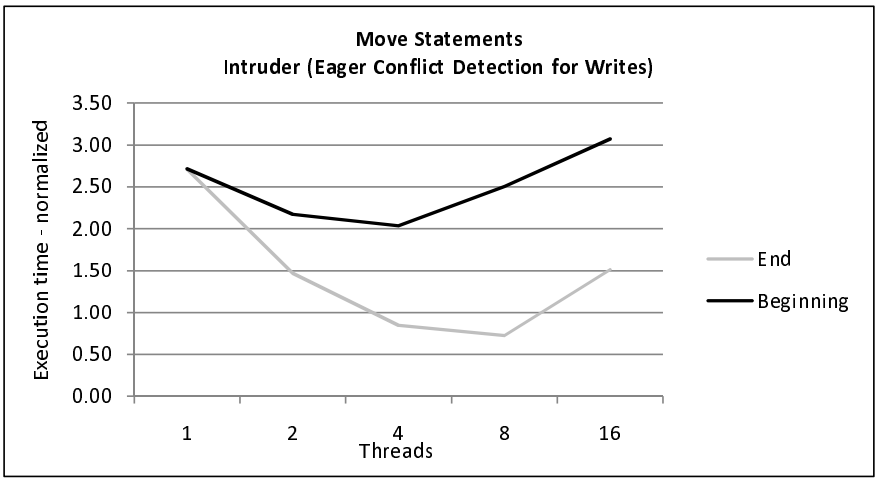


Figure 6.14: This figure shows the effect of changing the location of only one statement inside an `atomic` block on typical STM systems which detect Write-After-Write conflicts eagerly. At *Beginning* an update operation is near the beginning of an `atomic` block and at *End* the update operation is near the end of the `atomic` block.

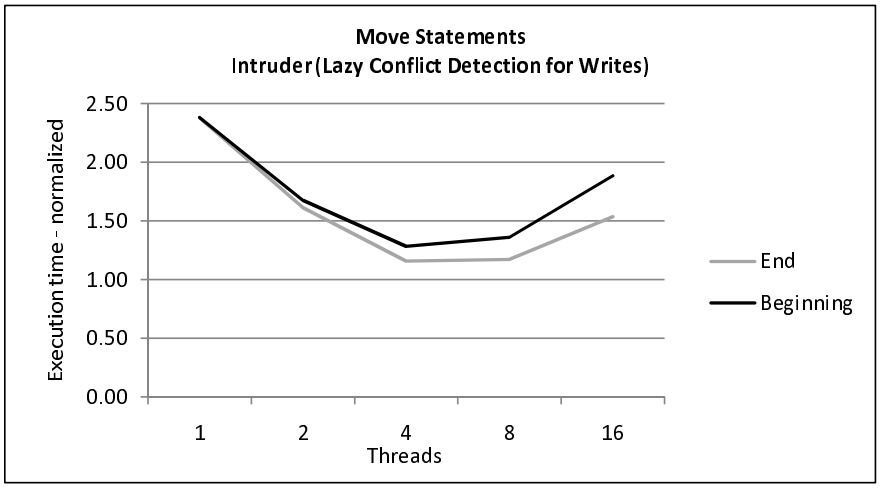


Figure 6.15: This figure shows the effect of changing the location of only one statement inside an `atomic` block on typical STM systems which detect Write-After-Write conflicts lazily. At *Beginning* an update operation is near the beginning of an `atomic` block and at *End* the update operation is near the end of the `atomic` block.

#Thd	Version	InsertSorted	Push	Commit	Abort	#Re-exec.	Wasted Work
2	Base	2.94%	48.06%	49.00%	1.88%	0.02	2.28%
	Push Move Up	0.00%	100.00%	0.00%	58.48%	1.43	40.16%
	Nested Push	11.77%	9.22%	79.01%	1.42%	0.01	1.14%
	Nested Insert	60.80%	38.79%	0.37%	1.38%	0.01	2.02%
	Nested Ins.+Push	98.54%	0.70%	0.76%	1.38%	0.01	1.36%
4	Base	19.16%	23.41%	57.42%	13.78%	0.16	14.74%
	Push Move Up	0.0%	100.00%	0.0%	70.60%	2.41	63.30%
	Nested Push	28.26%	2.01%	69.73%	9.50%	0.11	10.60%
	Nested Insert	88.15%	10.33%	1.52%	18.48%	0.27	14.36%
	Nested Ins.+Push	97.53%	0.08%	2.39%	8.27%	0.09	11.30%
8	Base	38.38%	13.31%	48.31%	36.16%	0.57	40.84%
	Push Move Up	0.00%	100.00%	0.0%	77.10%	3.38	83.45%
	Nested Push	44.13%	0.11%	55.76%	28.46%	0.40	42.02%
	Nested Insert	90.32%	1.50%	8.18%	13.45%	0.16	23.83%
	Nested Ins.+Push	99.05%	0.04%	0.91%	25.40%	0.34	39.60%

Table 6.4: The transactional characteristics of the `atomic` block which executes function `Decoder.Process` from Figure 6.13. *InsertSorted*, *Push* and *Commit* indicate the wasted work caused by the conflicts detected respectively at the calls to methods `InsertSorted` (line 9), `Push` (line 31) and when transaction commits (line 39). *Abort* indicates the abort rate of this `atomic` block. *#Re-execute* indicates the number of consecutive re-executions when abort happens. *Wasted Work* indicates the part of this `atomic` block execution which was wasted because of aborts.

#Thrd	Version	Norm. Time	Abort	WW
2	Base	0.54	2.38%	3.20%
	Push Move Up	0.80	28.35%	27.34%
	Nested Push	0.54	3.14%	2.66%
	Nested Insert	0.54	2.76%	3.30%
	Nested Ins. + Push	0.54	3.08%	2.98%
4	Base	0.31	11.52%	17.56%
	Push Move Up	0.75	72.22%	77.90%
	Nested Push	0.30	12.64%	16.10%
	Nested Insert	0.31	10.98%	18.48%
	Nested Ins. + Push	0.32	9.93%	15.77%
8	Base	0.27	32.12%	45.50%
	Push Move Up	0.92	90.10%	96.03%
	Nested Push	0.28	33.40%	53.48%
	Nested Insert	0.25	26.45%	36.80%
	Nested Ins. + Push	0.30	29.78%	47.38%

Table 6.5: Transactional characteristics of Intruder summarized for the whole program execution. *Norm. Time* is the normalized execution time of each version to its single threaded execution, *Abort* is the abort rate, *WW* is the wasted work caused by aborts.

Nested Push), 2) we have wrapped the call to `InsertSorted` in a nested `atomic` block (Table 6.4 version Nested Insert) and 3) we have wrapped both calls in nested `atomic` blocks (Table 6.4 version Nested Ins. + Push). We have extended Bartok-STM to support partial roll back for nested transactions i.e. if the outer transaction is valid, only the nested transaction will re-execute.

From conflict point discovery we can see that invoking `Push` inside a nested transaction reduces the wasted work and improves the performance of the outer `atomic` block (Table 6.4 version Nested Push). The nested `atomic` saves time by preventing the outer transaction from rollback and re-execution when it is valid. This modification has also changed the balance over the sources of wasted work by shifting some of the wasted work to `InsertSorted` and `Commit`. When only `InsertSorted` is wrapped in a nested `atomic` block we can see that the wasted work at the call to `InsertSorted` increases with the same amount at which conflicts on `Commit` decrease. This suggests that besides the WaW conflicts, there are also RaW and WaR conflicts which are detected at the end of the commit. When using nested transactions, most of these conflicts are detected when the nested transaction commits, otherwise the same conflicts are detected when the outer transaction commits. In other words, the nested `atomic` block changes the conflict detection to an earlier point during the execution of the outer `atomic` block (i.e. the end of the nested `atomic` block). In effect, this reduces the amount of speculative execution due to conflicts which otherwise would be discovered at the end of the outer `atomic` block. Using nested `atomic` blocks at both places subsumes the observed results from conflict point discovery (Table 6.4 version Nested Ins. + Push).

Table 6.5 shows the summarized results over the whole program execution for the different versions of Intruder. These results suggest that the best performance for 4 threads is achieved when `Push` is called inside a nested `atomic` and for 8 threads when `InsertSorted` is called inside nested `atomic` block. Despite the lower wasted work the execution time of Intruder is not significantly better than the base version. The reason is that nested `atomic` blocks incur small runtime overhead which is not always amortized by the saved wasted work.

Early release, which is demonstrated in the following section, is another technique that can squiz a bit more performance from Intruder. As described in

Figure 6.3, it is possible to use early release when packet segments are inserted in sorted order in `fragmentedListPtr` (Figure 6.13 line 9).

Last but not least, we would like to note that the authors of STAMP have designed this benchmark suite with the purpose to benchmark the performance of different TM implementations. Therefore, to benchmark broad spectrum of implementations it is not necessary that applications in this suite are implemented in the most optimal way and expected to scale. In fact, Intruder is a very useful workload because it illustrates how an application's behavior can be dependent on the TM system that it uses. We also believe that STAMP authors were aware that using hashtable instead of red black tree would make the application more scalable for STMs.

6.5.4 Labyrinth

Labyrinth implements a variant of Lee's path routing algorithm used in drawing circuit blueprints. The only data structure causing conflicts in this application was the grid on which the paths are routed. Almost all conflicts were happening in the method that copies the shared grid into a thread local memory. The wasted work due to the aborts at this place amounted to 80% of the total program execution with 2 threads and 98% with 4 threads. In this case we followed a well known optimization strategy described by Watson *et al.* [125]. The optimization is based on domain specific knowledge that the program still produces correct result even if threads operate on an outdated copy of the grid. Therefore, we annotated the `grid_copy` method to instruct the compiler to not instrument the memory accesses inside `grid_copy` with calls to the STM library, which in fact is functionally the same as using early release. After this optimization Labyrinth's execution was similar to the one reported by the STAMP suite's authors [20] (see Table 6.1).

Although our prior knowledge of the existing optimization technique, this use case serves as a good example when TM applications can be optimized by giving hints to the TM system in similar way as with early release.

#Threads	Application	Abort	Wasted Work
2	Genome	0.10%	0.10%
	Vacation	0.80%	1.20%
	WormBench	0.00%	0.00%
4	Genome	0.50%	0.20%
	Vacation	2.45%	4.80%
	WormBench	0.01%	0.02%
8	Genome	0.82%	0.50%
	Vacation	5.30%	7.90%
	WormBench	0.03%	0.07%

Table 6.6: Percentage of the wasted work due to aborts in Genome, Vacation and WormBench.

6.5.5 Vacation and WormBench

Vacation and WormBench scaled as reported by their respective authors and had very little wasted work (see Table 6.6). In these applications, there was not any opportunity for further optimizations.

In Vacation we saw that the most aborting `atomic` block encloses a while loop. We were tempted to move the `atomic` block inside the loop as in Figure 5.1 but that would change the specification of the application that the user can specify the number of the tasks to be executed atomically. Moving the `atomic` block inside the loop would always execute one task and therefore reduce the conflict rate but the user will no longer be able to specify the number of the tasks that should execute atomically. Also, similar changes may not always preserve the correctness of the program because they may introduce atomicity violation errors. In Genome, though very few, aborts occurred in the first and the last `atomic` blocks in the program order (see Figure 5.10). In our setup, WormBench had almost not conflicts — in 8-threaded execution from 400 000 transactions only about 1100 aborted.

6.6 Summary

In this chapter we have introduced techniques for optimizing transactional memory applications. These techniques are to be used in a methodical approach for optimizing applications to a specific TM implementation when profiling information is available. We have profiled the applications from the STAMP TM benchmark suite using a TM-enabled profiling tool. Although these applications are carefully written to have minimal overheads, we could find a performance niche in them which we used to demonstrate the effect of our optimization techniques.

In Genome an open addressing hashtable was a congestion point of false conflicts due to the object level conflict detection in our STM library. It was easy to see this problem using conflict point discovery. A solution was to replace the open addressing hashtable with a chaining hashtable in which elements are stored in a separate linked list object thus eliminating the false conflicts.

In Bayes, after a simple profiling, we have seen that only two mutually aborting `atomic` blocks are responsible for almost all the wasted work in the program. We have statically scheduled these two `atomic` blocks to not execute in parallel. Although our schedule introduced new aborts between other pair of `atomic` blocks, it decreased the amount of the total wasted work.

In Intruder, depending on whether the underlying TM system detects WaW conflicts eagerly or lazily, the location of a memory assignment may have significant impact on the program performance. In TMs which detect WaW conflicts eagerly, detecting such conflicts earlier during the execution of an `atomic` block can cause less wasted work but at the same time cause other threads executing the same `atomic` block to serialize. On the other side, TM systems which detect WaW conflicts lazily are not affected by the statements location. Also, in Intruder, we have shown that nested `atomic` blocks can be used as checkpoints. Checkpoints placed just before the conflicting statements can reduce wasted work on aborts and therefore improve the overall performance.

In Labyrinth we showed that early release can be a very effective way to reduce conflicts. However, yet this approach is not safe and should be applied carefully with knowledge about how the application algorithm can be relaxed.

In an iterative profile-and-optimize process, manual tuning can be automated with a feedback directed compilation. We have also discussed how our techniques can be implemented in feedback directed compilers.

Chapter 7

Conclusion

In this dissertation I studied the programmability aspects of real-world parallel programs using `atomic` blocks and transactional memory (TM). The goal set was to answer the following questions:

- *Is programming with* `atomic` *blocks and TM easier than locks?*
- *Is performance of TM competitive to locks?*
- *Is TM a mature technology to be used in developing production software?*

In Chapter 3, I have answered these questions by developing AtomicQuake. AtomicQuake was developed from the parallel version of the Quake game server by replacing all lock based synchronization with `atomic` blocks. The experience on developing AtomicQuake answered the above questions and also showed that the following assertions which are part of the thesis statement are true:

1. *Parallel programming using* `atomic` *blocks is easier than fine-grain locking schemes;*
2. *TM is not a bottleneck for the scalability of the parallel applications. However, unlike the performance results obtained with micro-benchmarks and small kernel applications, TM is not as efficient as locks in large real world applications. In a real transactional application TM has high single threaded overhead and unanticipated abort overheads at the presence of contention;*

3. *TM technology is not mature enough to be used for developing production software because of the following reasons:*

   (a) *Language extensions and semantics are not expressive enough to implement I/O, errors and recover from errors inside transactions. For example, locks cannot be replaced directly because their use do not match the block based structure of* `atomic` *blocks;*

   (b) *Existing application development tools such as compilers, debuggers and profilers have minimal or no support for TM. For example, debuggers are not aware of* `atomic` *blocks and they cannot execute* `atomic` *blocks atomically. Also, existing profiling tools do not provide relevant information to discover and understand the TM overheads;*

4. *In large parallel applications replacing the lock-based synchronization with* `atomic` *blocks is not straightforward. It requires careful examination of the code to understand the locking policy (i.e. which lock protects which shared data).*

AtomicQuake contains rich uses of transactions and also its transactions have different runtime characteristics. All these make AtomicQuake a valuable workload for benchmarks complete implementations of TM which span across several layers on the software stack. It is an important contribution because it has driven the research in TM further by opening new problems. While the other researchers were busy with addressing some of these problems I have focused on three of them:

1. Debugging support for TM applications;

2. Profiling techniques for TM applications;

3. Optimization approaches for TM applications.

In Chapter 4 I have investigated how to extend existing debuggers with support for programs that use `atomic` blocks and TM. I have contributed to the TM research by introducing new debugging techniques that fall in three categories:

1. debugging at the level of atomic blocks;

2. debugging at the level of transactions;

3. managing transactions at debug-time.

When debugging at the level of `atomic` blocks a whole block is executed as if it was a single instruction. If the user wants to step inside an atomic block to debug wrong code the debugger ensures that the user will not observe inconsistent speculative values and aborts. This approach extends the debugger with the atomicity and isolation semantics of transactions. It is intuitive and also abstracts the underlying implementation of `atomic` blocks which might be either based on lock inference such as a global lock or TM. Conversely, when debugging at the level of transactions the implementation of `atomic` blocks is not any more abstracted and the programmer can observe the state of the underlying TM implementations. This approach is mainly intended for debugging performance errors - for instance identifying the instructions which are responsible for a conflict. To debug synchronization errors I have introduced a new debugger abstraction – debug-time transaction. Debug-time transactions let the programmer to manipulate the synchronization by introducing new `atomic` blocks or enlarging the scope of existing `atomic` blocks from within the debugger. I have implemented these ideas in a debugger extension for WinDbg and a debugging framework for Bartok-STM. Findings in Chapter 4 showed that the following assertion which is part of the thesis statement is true:

5. *It is difficult to find synchronization errors in TM applications and debug wrong code inside* `atomic` *blocks because conventional debuggers are not aware of* `atomic` *blocks and TM. To find the synchronization errors between* `atomic` *blocks such as atomicity violations and asymmetric data races debuggers need to be extended with the atomicity semantics of transactions. To debug wrong code inside* `atomic` *blocks without observing speculative updates from other transactions, debuggers need to be extended with the isolation semantics of transactions;*

In Chapter 5 I have investigated methods to profile transactional applications and report the profiling results in a form independent from the underlying TM implementation. I have contributed to the TM research by introducing new a series

of profiling techniques for transactional applications. These techniques provide in depth and comprehensive information to help the programmer in identifying and understanding the bottlenecks specific to the TM programming model. These techniques can be classified in three categories:

1. techniques to identify multiple conflicting locations;
2. techniques to identify conflicting objects; and
3. techniques to visualize how threads spend their time and how transactions progress.

I have demonstrated that these techniques can be implemented efficiently for an existing STM – Bartok-STM. I have extended Bartok-STM with a lightweight profiling framework which is responsible to collect runtime data. To minimize the overhead and probe effect, the large portion of the raw data is processed offline by a visualization tool and the remaining small portion of data is processed at runtime during during garbage collection. Findings in Chapter 5 showed that the following assertion which is part of the thesis statement is true:

6. *TM applications have different types of bottlenecks which are specific to the TM programming model. These bottlenecks are caused by the aborting transactions and are difficult to anticipate and understand. To find and understand these bottlenecks properly requires new profiling techniques which report results in an from independent of the underlying TM implementation;*

In Chapter 6 I have examined the TM specific bottlenecks by profiling applications from STAMP TM benchmark suite and from the synthetic WormBench workload. I have contributed to the TM research by introducing series of techniques for a methodical optimization of transactional applications. The target of these optimization techniques is to reduce the wasted work caused by aborting transactions. They require knowledge of the underlying TM implementation and leverage low-level mechanisms. I have demonstrated the effectiveness of these techniques by optimizing Genome, Bayes, Intruder and Labyrinth applications

from the STAMP TM benchmark suite. Findings in Chapter 6 showed that the last assertion which is part of the thesis statement is true:

7. *The performance of TM applications can be improved with TM-specific optimizations which leverage the specific mechanisms provided by the underlying TM implementation. For example, the same program can execute faster if the programmer uses transaction checkpoints, nested* `atomic` *blocks or early release.*

7.1 Future Work

Research described in this dissertation can be extended in two directions. The first approach is to investigate in more depth feedback directed optimizations and their implementation. This would naturally follow from the profiling and optimization work described in Chapter 5 and Chapter 6 respectively. While working on the profiling and optimization techniques for transactional applications I have noticed that there is a potential for improving the application performance by giving compile-time hints to the TM system. For example, the programmer can statically schedule the execution of two `atomic` blocks which abort each other to not overlap, or the programmer can statically choose between lazy and eager conflict detection for a given object, or the programmer can statically change the granularity of conflict detection for a specific object which is involved in false conflicts. Some of these ideas are already explored in runtime implementations; however runtime implementations require additional booking and incur overheads. Based on the profiling information the runtime characteristics of the STAMP applications are regular and do not change in time suggesting that static settings would suffice. It would be interesting to implement these ideas in the Bartok-STM compiler and compare their performance with the respective runtime implementations (for those which exist). Also, it would be interesting to study hybrid implementations such as combination between static and runtime, where the static hints are used to offload the runtime overhead.

The second approach for extending this work is to carefully study the L1 data cache miss rate caused by the STM operations which manage the transactional

metadata (i.e. read and write set, logs, validation). While I was profiling the transactional applications from the STAMP benchmark suite I have noticed that the STM operations cause x3-x4 times more cache misses in a single-threaded program execution and the cache miss rate increase with the number of threads. This increase can be explained with the increase of the number of of instructions because of the STM operations. However, a smart hardware support for STM can reduce this cache miss rate. A reduced cache miss rate would improve the application performance by reducing the single threaded overhead of the STM and being less scalability bottleneck. Also, such hardware extension could be applied in other domains as well.

References

[1] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 185–196, Febrary 2009. 35, 50, 52

[2] Ahmed Abdelkhalek and Angelos Bilas. Parallelization and performance of interactive multiplayer game servers. In *IPDPS '04: Proc. 18th international parallel and distributed processing symposium*, pages 72–81, April 2004. 10, 47, 52, 55, 56, 59

[3] Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos. Behavior and performance of interactive multi-player game servers. In *Proc. Int. IEEE Symposium on the Performance Analysis of Systems and Software (ISPASS-2001*, pages 355–366, November 2001. 56

[4] Ali-Reza Adl-Tabatabai and Tatiana Spheisman. Draft specification of transactional langauge constructs for c++ version 1.0.2. Mainling group, April 2010. 20

[5] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, June 2006. 38, 155

[6] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. In *MSPC '06: Proc. 2006 Workshop on Memory System Performance and Correctness*, pages 70–81, March 2006. 29

[7] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. In *PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 151–162, February 2009. 29

[8] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCS '05: Proc. 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, February 2005. 45

[9] Mohammad Ansari, Kim Jarvis, Christos Kotselidis, Mikel Lujan, Chris Kirkham, and Ian Watson. Profiling transactional memory applications. In *PDP '09: Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 11–20, 2009. 148

[10] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The opentm transactional application programming interface. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 376–387, September 2007. 82

[11] Alexandro Baldassin and Sebastian Burckhardt. Lightweight software transactions for games. In *HotPar '09: HotPar 09: Proc. 1st Workshop on Hot Topics in Parallism*, March 2009. 51

[12] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *ISCA '08: Proc. 35th international symposium on computer architecture*, pages 115–126, June 2008. 52

[13] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987. ISBN 0-20110-715-5. 84

[14] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *WDDD '05: Proc. 4th workshop on duplicating, deconstructing and debunking*, pages 48–55, June 2005. 52

[15] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *WDDD '05: Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2005. 34

[16] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report TR-CIS-06-09, University of Pennsylvania, Department of Computer and Information Science, May 2006. 52

[17] Colin Blundell, Arun Raghavan, and Milo M.K. Martin. RETCON: transactional repair without replay. In *ISCA '10: Proc. 37th International Symposium on Computer Architecture*, pages 258–269, June 2010. 159

[18] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *ISCA '07: Proc. 34th international symposium on computer architecture*, pages 81–91, June 2007. 106, 121, 130, 173

[19] Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. Feedback-directed barrier optimization in a strongly isolated STM. In *POPL '09: Proc. of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 213–225, January 2009. 155

[20] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. 11th IEEE International Symposium on Workload Characterization*, pages 35–46, September 2008. 49, 102, 153, 157, 159, 162, 166, 168, 169, 175, 183

[21] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *OOPSLA '94: Proc. 9th annual conference on object-oriented programming systems, language, and applications*, pages 414–426, October 1994. 48

[22] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The atomos transactional programming language. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–13, June 2006. 35

[23] Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, JaeWoong Chung, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. TAPE: A transactional application profiling environment. In *ICS '05: Proc. 19th international conference on supercomputing*, pages 199–208, June 2005. 109, 132, 148, 164

[24] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A scalable, non-blocking approach to transactional memory. In *HPCA '07: Proc. 13th IEEE International Symposium on High Performance Computer Architecture*, pages 97–108, February 2007. 28, 44, 109, 176

[25] Dhruva R. Chakrabarti. New abstractions for effective performance analysis of STM programs. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 333–334, January 2010. 15, 133, 143, 153, 158, 159, 166

[26] Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. Using profile information to assist classic code optimizations. *Softw. Pract. Exper.*, 21 (12):1301–1321, 1991. ISSN 0038-0644. 164

[27] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *PLDI '08: Proc. International Conference on Programming Language Design and Implementation*, pages 304–315, June 2008. 24

[28] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *ASPLOS '06: Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–358, October 2006. 45

[29] Jaewoong Chung, Hassan Chafi, Chi Cao Minh, Austen Mcdonald, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. The common case transactional behavior of multithreaded programs. In *HPCA '06: Proc. 12th International Conference on High-Performance Computer Architecture*, pages 266–277, February 2006. 48

[30] Intel Corporation. *Intel® C++ STM Compiler Prototype Edition 2.0 Language Extensions and User's Guide*. Intel Corporation, 2 edition, March 2008. 61, 77

[31] David Cunningham, Khilan Gudka, and Susan Eisenbach. Keep off the grass: Locking the right path for atomicity. In *CC '08: Proc. International Conference on Compiler Construction*, pages 276–290, April 2008. 24

[32] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *ASPLOS '06: Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, October 2006. 45

[33] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proc. 20th ACM International Symposium on Distributed Computing*, pages 194–208, September 2006. 24, 38, 158, 166, 169

[34] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, March 2009. 49, 133, 156

[35] Shlomi Dolev, Danny Hendler, and Adi Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *PODC '08: Proc. of 27th ACM Symposium on Principles of Distributed Computing*, pages 125–134, August 2008. 33, 156

[36] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Dividing transactional memories by zero. In *TRANSACT '08: 3rd Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, February 2008. 49

[37] Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *PODC '09: Proc. of the 28th ACM Symposium on Principles of Distributed Computing*, pages 7–16, 2009. 33, 156

[38] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR–06–052, Intel, 2006. 160

[39] Pascal Felber, Christof Fetzer, Ulrich Mller, Torvald Riegel, Martin Skraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT '07: 2nd Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, August 2007. 82, 153, 157, 164

[40] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *DISC '09: Proc. 23rd international Conference on Distributed Computing*, pages 93–107, September 2009. 37

[41] Keir Fraser. *Practical lock freedom.* PhD thesis, PhD Thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579. 35

[42] Vladimir Gajinov, Ferad Zyulkyarov, Osman S. Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. Quaketm: parallelizing a complex sequential application using transactional memory. In *ICS '09: Proc. 23rd international conference on Supercomputing*, pages 126–135, June 2009. 8, 11, 14, 20, 50, 105, 118, 130, 135, 144

[43] Dan Grossman. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proc. 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 695–706, October 2007. 19

[44] Andrew S. Grove. *Only the paranoid survive*. Doubleday, 1st edition, 1996. ISBN 978-0385482585. 1

[45] Rachid Guerraoui and Michal Kapalka. On obstruction-free transactions. In *SPAA '08: Proc. of the 20th Symposium on Parallelism in Algorithms and Architectures*, pages 304–313, June 2008. 160

[46] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A benchmark for software transactional memory. In *EuroSys '07: Proc. 2nd European systems conference*, pages 315–324, March 2007. 12, 48

[47] Shantanu Gupta, Florin Sultan, Srihari Cadambi, Franjo Ivancic, and Martin Rotteler. Using hardware transactional memory for data race detection. In *IPDPS '09: Proc. 23rd IEEE international parallel and distributed processing symposium*, pages 1–11, May 2009. 109, 124

[48] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *PACT '07: Proc. 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 353–364, September 2007. 24

[49] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Michael Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. In *MICRO '04: Proc. 37th IEEE/ACM International Symposium in Microarchitecture*, pages 92–103, November 2004. 28, 44

[50] Derin Harmanci, Vincent Gramoli, Pascal Felber, and Christof Fetzer. Extensible transactional memory testbed. *JPDC '10: Journal of Parallel Distributed Computcomputing*, 70:1053–1067, October 2010. 49

[51] Tim Harris. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58:325–343, December 2005. ISSN 0167-6423. 35

[52] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proc. 18th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, pages 388–402, October 2003. 105, 144, 153

[53] Tim Harris and Srdjan Stipic. Abstract nested transactions. In *TRANSACT '07: 2nd Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, August 2007. 32, 165

[54] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, February 2005. 21, 68, 160

[55] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25, June 2006. 38, 40, 82, 95, 109, 131, 149, 153, 155, 157, 164, 166

[56] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2nd edition, June 2010. ISBN 978-1598291247. 2, 38, 72, 105, 134

[57] Tim Harris, Saša Tomic, Adrián Cristal, and Osman Unsal. Dynamic filtering: multi-purpose architecture support for language runtime systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 39–52, March 2010. 45

[58] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proc. 13th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 207–216, February 2008. 52

[59] Maurice Herlihy and Yossi Lev. tm_db: A generic debugging library for transactional programs. In *PACT '09: Proc. 18th international conference on parallel architectures and compilation techniques*, pages 136–145, September 2009. 13, 108, 111, 121

[60] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proc. 20th International Symposium on Computer Architecture*, pages 289–300, May 1993. 2, 24, 44, 48

[61] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proc. 22nd Symposium on Principles of Distributed Computing*, pages 92–101, July 2003. 35

[62] Michael Hicks, Jeffrey S. Foster, and Polyvios Prattikakis. Lock inference for atomic sections. In *TRANSACT '06: 1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006. 24

[63] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, pages 1170–1183, 1986. 1

[64] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Eigenbench: A simple exploration tool for orthogonal tm characteristics. In *IISWC '10: 2010 International Symposium on Workload Characterization*, pages 1–11, December 2010. 12, 49

[65] ID Software. Quake. 50, 55

[66] Seunghwa Kang and David A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 15–24, Febryary 2009. 49, 156

[67] Gokcen Kestor, Vasileios Karakostas, Osman Unsal, Adrian Cristal, Ibrahim Hur, and Mateo Valero. RMS-TMS: A transactional memory benchmark for recognition, mining and synthesis applications. In *ICPE '11: Proc. International Conference on Performance Engineering*, March 2011. 50, 105, 118, 144

[68] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *PPoPP '06: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 209–220, March 2006. 45

[69] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proc. of 2004 International Symposium on Code Generation and Optimization*, pages 75–86, March 2004. 164

[70] Suh-Yin Lee and Ruey-Long Liou. A multi-granularity locking model for concurrency control in object-oriented database systems. In *Transactions on Knowledge and Data Engineering*, pages 144–156, February 1996. 60, 85

[71] Yossi Lev. Making debuggers transaction-ready. Transactional Memory: From Implementation to Application, Seminar 2008241, Dagstuhl, Germany, June 2008. 111

[72] Yossi Lev and Jan-Willem Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *PPoPP '08: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, February 2008. 35

[73] Yossi Lev and Mark Moir. Debugging with transactional memory. In *TRANSACT' 07: 2nd Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, August 2007. 13, 109

[74] Yossi Lev, Victor Luchangco, Virendra J. Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional

memory. In *TRANSACT' 09: 4th Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, February 2009. 109

[75] Yi Liu, Xin Zhang, He Li, Mingxiu Li, and Depei Qian. Hardware transactional memory supporting i/o operations within transactions. In *HPCC '08: Proc. 10th IEEE International Conference on High Performance Computing and Communications*, pages 85–92, September 2008. 35, 68

[76] Joao Lourenço, Ricardo Dias, Joao Luís, Miguel Rebelo, and Vasco Pessanha. Understanding the behavior of transactional memory applications. In *PADTAD '09: Proceedings of the 7th Workshop on Parallel and Distributed Systems*, pages 1–9, July 2009. 16, 133, 153, 164

[77] Daniel Lupei, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *EuroSys: '10: Proc. 5th European Conference on Computer Systems*, pages 41–54, April 2010. 11, 51, 53, 156

[78] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *PPoPP '10: Proce. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–90, 2010. 33, 156

[79] Sandya Mannarswamy, Dhruva R. Chakrabarti, Kaushik Rajan, and Sujoy Saraswati. Compiler aided selective lock assignment for improving the performance of software transactional memory. In *PPoPP '10: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 37–46, January 2010. 27

[80] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–358, January 2006. 24

[81] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *ISCA '06: Proc. 33rd International Symposium on Computer Architecture*, pages 53–65, June 2006. 35

[82] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabi, Richard L. Hudson, Bartin Saha, and Adam Welc. Practical weak-atomicity semantics for Java STM. In *SPAA '08: Proc. 20th annual symposium on parallelism in algorithms and architectures*, pages 314–325, June 2008. 52

[83] Microsoft Corporation – MSDN. Debugger engine and extension APIs, . http://msdn.microsoft.com/en-us/library/cc267863.aspx. 109

[84] Microsoft Corporation – MSDN. Debugging tools for windows, . http://msdn.microsoft.com/en-us/library/cc266321.aspx. 109

[85] Milos Milovanovic, Osman S. Unsal, Adrián Cristal, Srdjan Stipic, Ferad Zyulkyarov, and Mateo Valero. Compile time support for using transactional memory in c/c++ applications. In *INTERACT '07: Proc. 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*, February 2007. 8

[86] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proc. 34th International Symposium on Computer architecture*, pages 69–80, June 2007. 9, 45

[87] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: log-based transactional memory. In *HPCA '06: Proc. 12th IEEE International Symposium on High Performance Computer Architecture*, pages 254–265, February 2006. 28, 44

[88] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proc. 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 68–78, March 2007. 29, 52

[89] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for C/C++. In *OOPSLA '08: Proc. 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 195–212, October 2008. 50, 82, 144, 153, 164

[90] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, pages 26–29, 2005. 1

[91] Zhelong Pan and Rudolf Eigenmann. Rating compiler optimizations for automatic performance tuning. In *SC '04: Proc. of the 2004 ACM/IEEE Conference on Supercomputing*, page 14, November 2004. 165

[92] Victor Pankratius, Ali-Reza Adl-Tabatabai, and Frank Otto. Does transactional memory keep its promises? Results from an empirical study. Technical Report 2009-12, University of Karlsruhe, September 2009. 20, 118, 130, 132

[93] Cristian Perfumo, Nehir Sonmez, Srdan Stipic, Adrian Cristal, Osman Unsal, Tim Harris, and Mateo Valero. The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. In *CF '08: Proc. ACM international conference on computing frontiers*, pages 67–78, May 2008. 50, 118, 144, 148

[94] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA '05: Proc. 32nd International Symposium on Computer Architecture*, pages 494–505, July 2005. 45

[95] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. MetaTM/TxLinux: transactional memory for an operating system. In *ISCA '07: Proc. 34th International Symposium on Computer Architecture*, pages 92–103, June 2007. 51

[96] Torvald Riegel, Pascal Felber, and Christof Fetzer. Dynamic performance tuning of word-based software transactional memory. In *PPoPP'08: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246, February 2008. 27, 38

[97] Michael F. Ringenburg and Dan Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP '05: Proc. 10th ACM SIGPLAN International Conference on Functional Programming*, pages 92–104, September 2005. 21

[98] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP '07: Proc. 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 87–102, October 2007. 51

[99] Christopher J. Rossbach, Hany E. Ramadan, Owen S. Hofmann, Donald E. Porter, Aditya Bhandari, and Emmett Witchel. TxLinux and MetaTM: transactional memory and the operating system. *Commununications of the ACM*, 51:83–91, September 2008. ISSN 0001-0782. 51

[100] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *PPoPP '10: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 47–56, January 2010. 20, 118, 130, 132

[101] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, March 2006. 38, 41

[102] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO '06: Proc. 39th IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, December 2006. 45

[103] Serap Savari and Cliff Young. Comparing and combining profiles. *The Journal of Instruction-Level Parallelism*, 2(1942-9525), May 2000. 166

[104] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proc. 20th ACM Symposium on Principles of Distributed Computing*, pages 240–248, July 2005. 28

[105] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proc. of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–88, June 2007. 35, 52

[106] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *ISCA '07: Proc. 34th International Symposium on Computer Architecture*, pages 104–115, June 2007. 45

[107] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *ISCA '08: Proc. 35th International Symposium on Computer Architecture*, pages 139–150, June 2008. 44

[108] Travis Skare and Christos Kozyrakis. Early release: Friend or foe? In *WTMW '06: In Workshop of Transactional Memory Workloads*, June 2006. 162

[109] Travis Skare and Christos Kozyrakis. Early release: Friend or foe? In *WTW '06: In 2006 Workshop of Transactional Memory Workloads*, June 2006. 35

[110] Nehir Sonmez, Cristian Perfumo, Srdjan Stipic, Adrian Cristal, Osman S. Unsal, and Mateo Valero. Unreadtvar: Extending haskell software transactional memory for performance. In *TFP '07: In Proc. 8th Symposium on Trends in Functional Programming*, April 2007. 35, 162

[111] Nehir Sonmez, Adrían Cristal, Osman S. Unsal, Tim Harris, and Mateo Valero. Profiling transactional memory applications on an atomic block basis: A haskell case study. In *MULTIPROG '09: Proc. 2dn workshop on Programmability issues for multi-core computers*, January 2009. 133, 148

[112] Nehir Sonmez, Tim Harris, Adrián Cristal, Osman Unsal, and Mateo Valero. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *IPDPS '09: Proc. 23rd IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, May 2009. 155, 166

[113] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *PODC'07: Proc. 26th ACM Symposium on principles of distributed computing*, pages 338–339. August 2007. 144

[114] Michael F. Spear, Luke Dalessandro, Virendra Marathe, and Michael L. Scott. Ordering-based semantics for software transactional memory. In *OPODIS '08: Proc. 12th International Conference on Principles of Distributed Systems*, pages 275–294, December 2008. 144

[115] Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and exploiting inevitability in software transactional memory. In *ICPP '08: Proc. 37th IEEE international conference on parallel processing*, pages 59–66, October 2008. 114, 120

[116] Michael F. Spear, Maged M. Michael, Michael L. Scott, and Peng Wu. Reducing memory ordering overheads in software transactional memory. In *CGO '09: Proc. 7th IEEE/ACM International Symposium on Code Generation and Optimization*, pages 13–24, March 2009. 35, 68

[117] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1994. ISBN 978-0132199087. 84

[118] Andrew S. Tanenbaum. *Modern Operating Systems (2nd Edition)*. Prentice Hall, 2001. ISBN 978-0130313584. 60, 85

[119] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. Eazyhtm: eager-lazy hardware transactional memory. In *MICRO '09: Proc. 42nd IEEE/ACM International Symposium on Microarchitecture*, pages 145–155, December 2009. 28, 44, 176

[120] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *CGO '03: Proc. of the 2003 International Symposium on Code Generation and Optimization*, pages 204–215, March 2003. 165

[121] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. xcalls: safe i/o in memory transactions. In *EuroSys '09: Proc. 4th ACM European Conference on Computer Systems*, pages 247–260, April 2009. 35

[122] M. M. Waliullah and Per Stenstrom. Intermediate checkpointing with conflicting access prediction in transactional memory systems. Technical report, Chalmers University of Technology. 159

[123] M. M. Waliullah and Per Stenstrom. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In *IPDPS '08: Proc. 22nd IEEE International Parallel & Distributed Processing Symposium*, pages 1–11, April 2008. 31

[124] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 34–48, March 2007. 61, 77, 82

[125] Ian Watson, Chris Kirkham, and Mikel Lujan. A study of a transactional parallel routing algorithm. In *PACT '07: Proc. 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 388–398, September 2007. 49, 162, 183

[126] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proc. 20th Symposium on Parallelism in Algorithms and Architectures*, pages 285–296, June 2008. 35, 66, 68, 114, 120

[127] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA '95: Proc. 22nd annual international symposium on computer architecture*, pages 24–38, June 1995. 50

[128] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA '07: Proc. 13th IEEE International Symposium on High Performance Computer Architecture*, pages 261–272, February 2007. 28, 44

[129] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA '08: Proc. of 20th Symposium on Parallelism in Algorithms and Architectures*, pages 169–178, 2008. 33, 156

[130] Richard M. Yoo, Yang Ni, Adam Welc, Bartin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *SPAA '08: Proc. 20th annual symposium on parallelism in algorithms and architectures*, pages 265–274, June 2008. 54, 66, 144, 162

[131] Craig Zilles and Lee Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *TRANSACT '06: 1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006. 35

[132] Ferad Zyulkyarov, Milos Milovanovic, Osman Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Memory management for transaction processing core in heterogeneous chip-multiprocessors. In *OSHMA '07: Workshop on Operating System support for Heterogeneous Multicore Architectures*, September 2007. 8

[133] Ferad Zyulkyarov, Osman S., Adrián Cristal, and Mateo Valero. Synthetic workloads for transactional memory. In *ACACES '07: Proc. 2nd Advanced Computer Architecture and Compilation for Embedded Systems*, pages 135–137, August 2007. 8

[134] Ferad Zyulkyarov, Sanja Cvijic, Osman Unsal, Adrián Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. Wormbench: a configurable workload for evaluating transactional memory systems. In *MEDEA '08: Proc. 9th workshop on MEmory performance*, pages 61–68, October 2008. 8, 12, 47, 105, 118, 166

[135] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *PPoPP '09: Proc. 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 25–34, February 2009. 8, 10, 20, 47, 85, 105, 107, 118, 144

[136] Ferad Zyulkyarov, Tim Harris, Osman S. Unsal, Adrían Cristal, and Mateo Valero. Debugging programs that use atomic blocks and transactional memory. In *PPoPP '10: Proc. 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, pages 57–66, January 2010. 7, 15

[137] Ferad Zyulkyarov, Srdjan Stipic, Tim Harris, Osman S. Unsal, Adrían Cristal, Ibrahim Hur, and Mateo Valero. Discovering and understanding performance bottlenecks in transactional applications. In *PACT'10: Proc. 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 285–294, September 2010. 7, 15, 153, 164

[138] Ferad Zyulkyarov, Srdjan Stipic, Tim Harris, Osman S. Unsal, Adrían Cristal, Ibrahim Hur, and Mateo Valero. Discovering and understanding performance bottlenecks in transactional applications. In *IJPP'11: to appar at International Journal of Parallel Programming*, 2011. 7