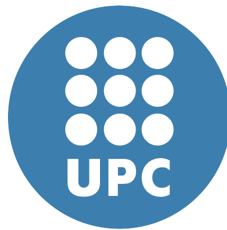
ADVERTIMENT. La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del servei TDX (www.tesisenxarxa.net) ha estat autoritzada pels titulars dels drets de propietat intel·lectual únicament per a usos privats emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA. La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del servicio TDR (www.tesisenred.net) ha sido autorizada por los titulares de los derechos de propiedad intelectual únicamente para usos privados enmarcados en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio TDR. No se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING. On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the TDX (www.tesisenxarxa.net) service has been authorized by the titular of the intellectual property rights only for private uses placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading and availability from a site foreign to the TDX service. Introducing its content in a window or frame foreign to the TDX service is not authorized (framing). This rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author

Enhancing the Efficiency and Practicality of Software Transactional Memory on Massively Multithreaded Systems



Gökçen Kestor

Department of Computer Architecture

Universitat Politècnica de Catalunya

A thesis submitted for the degree of
Doctor of Philosophy

February, 2013

Yesterday is history. Tomorrow is a mystery. Today is a gift. That's why we call it "The Present".

— **Alice Morse Earle**

Our greatest glory is not in never falling, but in rising every time we fall..

— **Confucius**

Abstract

Chip Multithreading (CMT) processors promise to deliver higher performance by running more than one stream of instructions in parallel rather than by increasing the processor's frequency. CMT processors come with different architectures: Chip Multi-Processor (CMP), Simultaneous Multi-Threading (SMT), or a combination of them. To exploit CMT's capabilities, users have to parallelize their applications. Unfortunately, the complexity of parallel programming and the difficulty of writing efficient and correct code limit the effective use of these systems.

Transactional Memory (TM) is one of programming models that aims at simplifying synchronization by raising the level of abstraction, breaking the connection between semantic atomicity and the means by which that atomicity is achieved. Programmers indicate atomic section in the source code without explicitly locking individual shared memory locations. An underlying TM system executes such transactions concurrently whenever possible, generally by means of speculation – optimistic but checked execution, roll-backing when conflicts arise. While TM is a promising programming model to simplify synchronization among parallel threads, there are still important challenges that must be addressed to make TM more practical and efficient in mainstream parallel programming. This dissertation presents work towards improving the practicality of TM across three dimensions.

The first challenge addressed is that of making the evaluation of TM proposals more solid with realistic TM benchmarks and being able to run the same benchmarks on different STM systems. As researchers work to develop robust, mature STM, it becomes increasingly important to be able to effectively and fairly compare STM designs with benchmarks that are representative of real-world applications. To address this challenge, we first

introduce a benchmark suite, RMS-TM, a comprehensive benchmark suite to evaluate hardware and software TM implementations. RMS-TM consists of seven applications from the Recognition, Mining and Synthesis (RMS) domain that are considered representative of future workloads for multi-core systems. RMS-TM features current TM research issues such as nesting, I/O and system calls inside transactions, while also providing a mix of short and long transactions with small/large read and write sets with low/medium/high contention rates. On the other hand, most STM systems were implemented as user-level libraries: the programmer was expected to manually instrument not only transaction boundaries, but also individual loads and stores within transactions. This library-based approach was adequate for early experiments with micro-benchmarks, but it becomes increasingly tedious and error prone for larger applications. The use of different library interfaces in different research groups has also made it difficult to share applications across groups, or to make reliable performance comparisons. To enable researches to perform an "apples-to-apples" comparison, we then develop a software layer that allows researchers to test the same applications with interchangeable STM back ends.

The second challenge addressed is that of enhancing performance and scalability of TM applications running on aggressive multi-core/multi-threaded processors. Performance and scalability of current TM designs do not always meet the programmer's expectation, especially at scale. This is especially true for STM designs, where the overhead of instrumentation and transactions' management severely limits application's performance at large scale. To overcome this limitation, we propose a new STM design, STM^2, based on an assisted execution model in which time-consuming TM operations are offloaded to *auxiliary threads* while *application threads* optimistically perform computation. Surprisingly, our results show that is often more convenient to use additional processing elements to support computation rather than performance computation: STM^2 provides, on average, speedups between 1.8x and 5.2x (and up to 12.8x) over state-of-the-art STM systems. Moreover, we notice that assisted-execution systems may show low processor utilization. In order to alleviate this problem and to increase the efficiency of STM^2, we

enriched STM^2 with a runtime mechanism that automatically and adaptively detects application and auxiliary threads' computing demands and dynamically partition hardware resources between the pair. In order to bias the allocation of hardware resources in favor of computing intensive application threads or overloaded auxiliary threads, we leverage the hardware thread prioritization mechanism implemented in POWER machines. This dynamic mechanism further improves STM^2's performance (up to 85% over the standard STM^2 design) and efficiency.

The third challenge addressed is that of defining a notion of what it means for a TM program to be correctly synchronized. Since TM has reached a maturity level and several STM and HTM implementations are available, it is important to provide debugging tools that automatically check the correctness of C/C++ TM programs. The current definition of transactional data race requires all transactions to be totally ordered "as if" serialized by a global lock, which limits scalability of TM designs. To remove this constraint, we first propose to relax the current definition of transactional data race to allow a higher level of concurrency. Based on this relaxed definition, we propose the first practical race detection algorithm for C/C++ applications, namely TRADE, and implement the corresponding race detection tool. Then, we introduce a new definition of transactional data race that is more intuitive, transparent to the underlying TM implementation, can be used for a broad set of C/C++ TM programs, enables a wide range of implementation techniques to be used. Based on this new definition, we proposed *T-Rex*, an efficient and scalable race detection tool for C/C++ TM applications. Using TRADE and *T-Rex*, we have discovered subtle transactional data races in widely-used STAMP applications which have not been reported in the past. Our experiments also show that *T-Rex* is order or magnitude faster than TRADE, which increase programmer productivity.

Acknowledgements

I would like to thank my advisors, Osman Unsal, Adrian Cristal and Mateo Valero, for their sincere help and support during my graduate studies. They are not only technically solid researchers but also kind advisors.

I would also like to thank other professors who have been greatly influential to me during my Ph.D study. First, I am truly grateful to Tim Harris and Serdar Tasiran who have helped and advised me during a very decisive part of this work. I am also sincerely thankful to Michael L. Scott who guided me in the dark and taught me a lot about research and working discipline.

Without the love and support of my family, this would have been a very hard journey. They have guided me with their priceless wisdom and have always prayed for me with their heartfelt love. I thank my sister, Gokben, for giving me strength when I felt hopeless and disappointed and my mother, Serpil, without whom I would not be the person that I am today.

I would also like to thank my great friend Gulay for many years of true friendship, and for believing in me and encouraging me in this winding road. We exchanged ideas about technical problems and shared a fun time while having awesome coffees. I only regret that I did not spend more time with her.

It is the luck of my lifetime to meet Ilker and Justine and to have them as my friends. Many thanks for loyal friendship and all the unforgettable times we have spent together.

My love, my heart, my husband Roberto. Ever since I met him, my life has changed to be better, happier and brighter. I promise that I will do whatever it takes to protect a blessed life of ours. Thank him so much to support me each step of the way during my graduate studies.

Contents

I Introduction **1**

1 Introduction **3**

1.1 The Difficulty of Parallel Programming 3
1.2 Transactional Memory and Challenges 4
1.3 Contributions 6
1.4 Organization 8
1.5 Publications 8

2 Transactional Memory **11**

2.1 TM Programming Model 11
2.1.1 Semantics 12
2.1.2 Programming 13
2.2 Implementation Options 16
2.2.1 Eager and Lazy Data Versioning 16
2.2.2 Eager and Lazy Conflict Detection 17
2.2.3 Software and Hardware 19
2.2.4 Commonly Used STMs 21

II Comprehensive Evaluation of TM Systems **25**

3 RMS-TM Benchmark Suite **27**

3.1 Introduction 27
3.2 The Transactification Process 29
3.2.1 Pre-Transactification Phase 29
3.2.1.1 Static Pre-Transactification 30

3.2.1.2 Dynamic Pre-Transactification 32
3.2.2 Transactification Phase 33
3.2.2.1 STM Implementation 33
3.2.2.2 HTM Implementations 34
3.3 RMS-TM Overview 35
3.4 Evaluation 36
3.4.1 Intel STM Results 36
3.4.1.1 Transactional Behavior 37
3.4.1.2 Performance Analysis 38
3.4.2 EazyHTM Results 40
3.4.2.1 Transactional Behavior 41
3.4.2.2 Performance Analysis 42
3.4.3 ScalableTCC Results 43
3.4.3.1 Transactional Behavior 43
3.4.4 Comparison of RMS-TM and STAMP 44
3.5 Related Work 46
3.6 Conclusions 48

4 Interchangeable Back Ends for STM Compilers 49

4.1 Introduction 49
4.2 Design and Implementation 51
4.2.1 Draft Specification for TM in C++ 51
4.2.2 Intel ABI Overview 52
4.2.3 Design Details 54
4.3 Experimental Setup 55
4.4 Experimental Results 57
4.4.1 Overhead Analysis of Automatic Instrumentation 57
4.4.2 Back-end Comparisons 59
4.5 Related Work 64
4.6 Conclusions 65

III Design and Implementation of a High Performance STM 67

5 STM^2: A Parallel STM for High Performance SMT Systems 69

5.1 Introduction 69
5.2 Motivation 72
5.3 STM^2 Design and Implementation 73
5.3.1 Application/Auxiliary Thread Synchronization 76
5.3.2 Writing to a shared memory location 77
5.3.3 Reading from a shared memory location 79
5.4 Experimental Setup 80
5.5 Experimental Results 81
5.6 Related Work 87
5.7 Conclusion 88

6 Enhancing the Performance of Assisted Execution Runtime Systems through Hardware/Software Techniques 91

6.1 Introduction 91
6.2 Hardware resource partitioning 93
6.3 Static Fine-Grained resource partitioning 95
6.3.1 Embarrassingly parallel phases 96
6.3.2 Load imbalance inside transactions 99
6.3.2.1 Overloaded application threads 100
6.3.2.2 Overloaded auxiliary threads 102
6.4 Adaptive Fine-Grained resource partitioning 106
6.5 Experimental results 112
6.5.1 Eigenbench 113
6.5.2 STAMP applications 114
6.6 Related work 119
6.7 Conclusions 120

IV Correctness Semantics for TM applications **123**

7 TRADE: Precise Dynamic Race Detection for Scalable Transactional Memory Systems **125**

7.1 Introduction 125
7.2 Background 128
7.3 Preliminaries 130
7.3.1 Strict Transactional Happens-Before Relation 130
7.3.2 Relaxed Transactional Happens-Before Relation 131
7.4 Transactional Race Detection Algorithms 135
7.4.1 s-TRADE Race Detection Algorithm 136
7.4.2 TRADE Race Detection Algorithm 138
7.4.3 Extensions 142
7.5 Design and Implementation 143
7.5.1 Binary instrumentation Framework 143
7.5.2 TRADE Instrumentation State and Code 143
7.6 Evaluation 146
7.7 Related Work 152
7.8 Conclusions 153

8 T-Rex: A Dynamic Race Detection Tool for C/C++ Transactional Memory Applications **155**

8.1 Introduction 155
8.2 Motivation 157
8.3 Preliminaries 160
8.4 Design and Implementation 164
8.4.1 Threads Data Access Table 164
8.4.2 Non-Transactional Memory Accesses 165
8.4.3 Transactional Memory Accesses 166
8.4.4 *T-Rex* Race Detection 167
8.5 Experimental results 170
8.5.1 *T-Rex* Race Detection Coverage 170
8.5.2 Overhead analysis 173
8.6 Conclusions 175

V Conclusions **177**

9 Conclusions **179**

A TRADE correctness proofs **185**

List of Figures

2.1 Lock-based vs. TM-based Programming 13
2.2 Nesting: The transaction within swap() is nested within the outer transaction by the program control flow 14
2.3 Example code for the use of "retry" primitive: If the retry statement is reached, the transaction is aborted and re-executed. 16
2.4 Type of Used Transactions in Programming Languages 19

3.1 Scalability of the lock-based applications, with the largest data sets, normalized to single-threaded lock execution time. 32
3.2 Scalability of the lock-based and TM-based applications, with the largest data sets, normalized to single-threaded lock and TM execution time, respectively, with Intel STM. 39
3.3 Runtime overhead of *TM-Fluidanimate* and *TM-UtilityMine* 40
3.4 Scalability of the lock-based and TM-Based applications normalized to single-threaded lock and TM execution time, respectively, using EazyHTM. 42
3.5 Scalability of the lock-based and TM-based applications normalized to single-threaded lock and TM execution time, respectively, with ScalableTCC. 44
3.6 Scalability of the STAMP applications normalized to single-threaded TM execution time. 45

4.1 Automatic read/write instrumentation of a simple TM program 53
4.2 Execution time of compiler-instrumented code, relative to manually instrumented code, for single-threaded STAMP applications. 58

4.3 Throughput results for the microbenchmarks. Y axis shows total number of transactions per second: higher is better. 60
4.4 Scalability results for STAMP and RMS-TM. 62

5.1 TinySTM per-transaction overhead breakdown for STAMP applications with respect to instrumented single thread version. 72
5.2 STM^2 offloads time-consuming STM operations to sibling hardware threads. 74
5.3 Pseudo-code for application and auxiliary thread STM write 78
5.4 Pseudo-code for application and auxiliary thread STM read 79
5.5 Performance comparison of different STMs with STAMP benchmarks. . 82
5.6 Speedups of STM^2over tested STMs for STAMP applications using the same amount of hardware resources (32 hardware threads). 83

6.1 Performance impact of reducing the priority of auxiliary threads when varying the percentage of time spent performing embarrassingly parallel computation and the value of Δ_p. 97
6.2 Frequently idle auxiliary threads within a transaction. 99
6.3 Performance impact of reducing the priority of auxiliary threads in presence of load imbalance within transactions (overloaded application threads).101
6.4 Execution trace of overloaded auxiliary threads. 102
6.5 Performance impact of increasing the priority of overloaded auxiliary threads when varying the number of read-set validations per transactional operation and Δ_p. 105
6.6 Irregular transactions with bursts of transactional operations. In this example Eigenbench executes a burst of transactional operations in the middle of the transaction. 106
6.7 The adaptive solution automatically changes the value of AxT_p according to the structure of the transaction and the computing demand of application and auxiliary threads. 111
6.8 Performance of static (best values among all combinations) and adaptive solutions for application with not-uniform transaction structure and varying size/position of burst of shared accesses. 114
6.9 Performance impact of static (best values among all combinations) and adaptive solutions for STAMP applications. 115

6.10 Labyrinth's transactions: alternation of a large local computation phase (white in the figure) with a burst of transactional operations (colored bars) at the end. 117

7.1 Does `ready=true` imply that `Thread 2` sees `data=42`? 129
7.2 Speedup of STAMP applications with various STMs. 132
7.3 This program has a transactional data race, as `Thread1` and `Thread2` may access x concurrently. 133
7.4 Example trace for the program in Figure 7.3 running on an STM that implements SGLA. No strict transactional data races detected. 138
7.5 Example trace for the program in Figure 7.3 running on an STM that does not implement SGLA semantics. HB detects a transactional data race in a given execution. 141
7.6 Implementation of the TRADE algorithm. 144
7.7 SSCA2 code snapshot. 147
7.8 TRADE runtime overhead over s-TRADE. 148

8.1 This program is intuitively racy but a race detection tool based on relaxed transactional data race definition produces different results according to thread interleaving: if `Thread 2` fully executes before `Thread 1` then the tool does not detect any transactional data races in a given execution. 158
8.2 Initially `shared = true` and `x = 0`. This program is intuitively correct but may result in incorrect behavior, depending on the underlying STM implementation. 159
8.3 *T-Rex* bookkeeping data structures: a) per-thread DAT; b) entry in the per-thread DAT. 165
8.4 *T-Rex* overall overhead and overhead breakdown for STAMP applications. 172
8.5 Comparing TRADE and *T-Rex* execution overhead over native execution with TL2. 174

List of Tables

3.1 Applications that pass the Static Pre-Transactification step. 30

3.2 Percentage of time spent inside critical sections with respect to total parallel time for the lock-based applications. The data sets used are appended to the application name. 31

3.3 Basic TM characteristics (with eight threads) of the RMS-TM applications, with Intel STM. 37

3.4 Percentage of time spent inside atomic blocks with respect to total parallel time for RMS-TM applications. 38

3.5 Configuration of the simulated system. 41

3.6 Transactional behavior of the RMS-TM applications with eight threads, with EazyHTM. 41

3.7 Transactional behavior of the RMS-TM applications with eight threads, with ScalableTCC. 43

4.1 Abort Rates (percentage of all dynamic transaction instances that abort) for 2, 4 and 8 threads. 59

6.1 Hardware thread priority levels in the IBM POWER7 processor. 94

7.1 Number of transactional data races detected by TRADE and s-TRADE without and with bug-injection. **Intruder* crashed because of the injected bug. 146

7.2 STAMP applications' characteristics. 150

7.3 Performance comparison between TRADE running on LLT and s-TRADE running on Pipeline. Time in seconds. 151

8.1 Number of detected transactional data races for STAMP applications for the original version and a version with synthetic bugs injected. *Intruder crashed because of the injected bug. 170
8.2 STAMP applications' characteristics. 173

Part I

Introduction

Multi-core systems have the potential for significant performance improvements, but the complexity of parallel programming and the difficulty of writing efficient and correct code limit the effective use of these systems. New programming models have been proposed to ease the development of parallel applications that perform well on multi-core architectures. Transactional Memory (TM) is one of such programming models that enables programmers to perform multiple memory operations atomically without worrying about the complexity issues associated with other programming models such as locks. Chapter 1 summarizes the thesis' contributions by highlighting the new research directions taken and the main results. Chapter 2 provides the necessary background on transactional memory: its definition, properties, implementation details and design trade-offs.

Chapter 1

Introduction

The performance of microprocessors has been continuously improving over the years thanks to advances in manufacturing technologies. In recent years, however, conventional techniques for improving single-threaded performance have begun hitting fundamental challenges such as the limited amount of instruction-level parallelism (ILP) [159] and the undesirable levels of power consumption caused by increasing clock frequencies [7].

In response, processor manufactures have shifted to Chip Multi-threading processors (CMTs) [56, 90, 107]. Multiple simpler processor cores in CMT systems promise to deliver higher performance by running more than one stream of instructions in parallel (thread-level parallelism (TLP)) in a power-efficient manner [124]. CMT processors may come with different architectures: Chip Multi-Processor (CMP), Simultaneous Multi-Threading (SMT), or a combination of them. With wide availability of CMTs, the burden of achieving scalable performance on CMTs has now been placed on programmers who must deal with the complexity of parallel programming to take the advantages of multiprocessors/multithreading.

1.1 The Difficulty of Parallel Programming

To increase parallelism on CMTs, programmers should create and synchronize several parallel tasks. For shared memory systems, the synchronization of parallel tasks is commonly handled by lock-level synchronization primitives. These primitives guarantee mutually exclusive shared memory accesses among all parallel tasks in the system.

Unfortunately, parallel programming with locks is quite difficult due to the trade-off between programming simplicity and scalability of the performance [72]. While adding coarse-grained locks to a program is relatively straightforward, it may drastically degrade performance since it introduces unnecessary serialization points during the execution. On the other hand, while fine-grained locking permits greater concurrency, its programming complexity is significantly higher, which even not result in a better performance than an equivalent coarse-grained version. The higher programming complexity may also cause various problems such as deadlock, convoying, or priority inversion [96].

1.2 Transactional Memory and Challenges

Transactional Memory [72] (TM) is a programming model to simplify synchronization by raising the level of abstraction, breaking the connection between semantic atomicity and the means by which that atomicity is achieved. Programmers indicate atomic sections in the source code (e.g., using language constructs such as *atomic* blocks, or using macros such as `BEGIN_TRANSACTION` and `END_TRANSACTION`) without explicitly locking individual shared memory locations. An underlying TM system executes such transactions concurrently whenever possible, generally by means of speculation—optimistic but checked execution, with rollback and retry when conflicts arise. There have been significant efforts to develop hardware (HTM) [42, 63], software (STM) [39, 43, 133] or hybrid TM systems [40, 91, 114].

While TM is a promising programming model to simplify the synchronization among parallel threads, there are still main important challenges that must be addressed to make TM more practical and efficient in mainstream parallel programming. First, realistic TM benchmarks and additional auxiliary software should be provided to make the evaluation of TM proposals more solid. Current benchmarks used to analyze the TM proposals do not include realistic applications that address ongoing TM research issues such as handling nested transactions, I/O operations, system and library calls inside transactions, and that provide the potential for straightforward comparison against locks. Thus, it is important to understand performance bottlenecks of TM on real applications and enable TM researchers to conduct their research by using changeling benchmarks. Moreover, most STM systems were implemented as user-level libraries:

the programmer was expected to manually instrument not only the transaction boundaries, but also individual loads and stores within the transactions. This library-based approach was adequate for early experiments with micro-benchmarks, but it becomes increasingly tedious and error prone for larger applications [37]. The use of different library interfaces in different research groups has also made it difficult to share applications across groups, or to make reliable performance comparisons: experiments with different versions of the application source code inevitably raise the questions of fairness and confidence. For C++ on the x86 architectures, significant steps in this direction have been made by compilers from Intel, the University of Dresden, and the GNU Project, which aim to accept the same application programming interface (API) and target the same runtime application binary interface (ABI). Unfortunately, these three compilers currently connect to only two main STM libraries (SkySTM [95] and TinySTM [133]). In the interest of greater interoperability, it is important to make works on STM systems compatible with recent compilers.

Second, it is essential to design a TM system for high performance, aggressive multi-threading systems. Most STM systems, so far, suffer from poor performance because the overhead introduced by the STM runtime system outweighs the performance gain achieved by the parallelism [24]. Some authors report drastic slow-downs when using STM (e.g., only breaking even with optimized sequential code after using 8 cores [24]). Even state-of-the-art TM systems typically require at least two threads to achieve performance that matches the performance of the optimized sequential code [39, 68]. To achieve the best possible performance on systems with the increasing number of cores/threads, it is significantly important to reduce STM runtime overhead and use all available resources effectively.

Third, TM should be supported with software development tools and integrated environment to help programmers debug and analyze TM applications. Perhaps the most important among these is race detection tools. A race condition occurs when a program's execution contains concurrent two accesses to the same memory location where at least one of the accesses is a write. Race conditions are particularly problematic because they typically cause problems only on certain rare interleavings, making them extremely difficult to detect, reproduce, and eliminate. Thus, it is crucial to support TM programmers by providing race detection tools.

1.3 Contributions

In this dissertation, we present studies conducted towards improving the efficiency and practicality of STM across three dimensions that we explain below. Specifically, this dissertation makes the following contributions:

1. **Comprehensive evaluation of TM systems**

   First, we introduce RMS-TM, a Transactional Memory benchmark suite composed of seven real-world applications from the Recognition, Mining and Synthesis (RMS) domain [86, 87]. In addition to featuring current TM research issues such as nesting and I/O and system calls inside transactions, the RMS-TM applications also provide a mix of short and long transactions with small/large read and write sets with low/medium/high contention rates. These characteristics, as well as providing lock-based versions of the applications, make RMS-TM a useful TM benchmark suite. Our experiments show that RMS-TM is scalable, which is useful for evaluating TM designs on high core counts. Second, to allow TM research groups to run each other's code and to perform apples-to-apples comparisons of implementation alternatives, we have implemented a "shim" library [81], which adapts the word-based "back end" libraries of the Rochester STM suite to the common ABI. This work makes the Rochester STM back ends available, for the first time, to programs written with language-level transactions. We also describe experience at both the ABI and API levels, and present performance comparisons relative to the Intel standard back end.

2. **Design and Implementation of a high performance STM**

   We have designed a novel parallel STM implementation, namely *Software Transactional Memory for Simultaneous Multi-threading* systems (STM^2) pronounced as *STM-squared* [82]. STM^2 reduces the runtime overheads by offloading read-set validation, bookkeeping, transaction state management and conflict detection to an *auxiliary thread* running on a *sibling* core/hardware thread, i.e., a processing element that shares some levels of hardware resource (like the L1 or L2 cache) with the *application thread*. Application threads optimistically perform their computation with minimal support from the underlying STM system. All synchronization and STM management operations are performed by the paired auxiliary threads.

This means that application threads experience minimal overhead. Auxiliary threads, instead, validate read-sets, maintain transaction states and detect conflicts in parallel with the application threads' computation. We exploit the fact that, on modern multi-core processors, sets of cores can share L1 or L2 caches. This lets us achieve closer coupling between the application thread and the auxiliary thread (when compared with a traditional multi-processor systems). We show that our approach outperforms several well-known STM implementations for various TM applications. In particular, STM^2 shows speedups between, on average, 1.8x and 5.2x over the tested STM systems with peaks up to 12.8x. Finally, we propose an approach to effectively partition processor resources between application and auxiliary threads in STM^2 [83, 84]. In order to bias the allocation of hardware resources in favor of computing intensive application threads or overloaded auxiliary threads, we leverage the hardware thread prioritization mechanism implemented in POWER machines. Our experiments show that effective hardware resource partitioning performs, in general, better than the original STM^2, up to 86% performance improvement.

3. **Providing Correctness Semantics for TM applications**

   We propose a novel and precise race detection algorithm for TM applications, namely TRADE that is based on a weakened definition of the happens-before relation and does not pose design constraints on the underlying STM system [88]. As a result, our algorithm can be used with a broader set of high-performance, scalable TM systems. Based on this definition, we implement a race detection tool for C/C++ TM applications. Our experiments reveal that TRADE precisely detects transactional data races. However, tools based on happens-before come with different kinds of issues such as high overhead, sensitivity to compiler and hardware optimizations and high dependency on the thread interleaving produced by the scheduler. In order to deal with those problems, we refine the definition of transactional data race and propose *T-Rex* [85]. *T-Rex* presents a new definition of transactional data race that follows the programmer's intuition of racy accesses, is independent of thread interleaving, can accommodate popular STM designs, and allows common programming idioms. We also compared *T-Rex* runtime

overhead to a race detection tool based on happens-before algorithm. Our results show that *T-Rex* is considerably faster than TRADE.

1.4 Organization

The rest of this thesis is organized as follows: Chapter 2 briefly reviews Transactional Memory. Chapter 3 presents RMS-TM benchmark suite and our methodology to choose candidate TM benchmarks from a set of real-world applications. Chapter 4 describes our "shim" library implementation which targets a fair comparison among several proposed STM systems. Chapter 5 describes the design of STM^2 and provides in-depth details of our current implementation. Moreover, it shows the performance numbers of STM^2 over tested STMs. Chapter 6 details our adaptive resource partitioning proposal with the POWER7 hardware thread priority mechanism and its impact on STM^2. Chapter 7 and Chapter 8 describes our transactional race detection algorithms based on a weakened definition of the happens-before relation and the definition that follows the programmers intuition of racy accesses, respectively. Finally, Chapter 9 concludes this dissertation.

1.5 Publications

1. G. Kestor, O. Unsal, A. Cristal and M. Valero, Transactifying Lock-Based RMS Applications, HiPEAC International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES), Poster Section, July, 2008, L'Aquila, Italy.

2. G. Kestor, S. Stipic, O. Unsal, A. Cristal and M. Valero, RMS-TM: A Transactional Memory Benchmark for Recognition, Mining and Synthesis Applications, The 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), February 2009, Raleigh, NC.

3. G. Kestor, V. Karakostas, O. Unsal, A. Cristal, I. Hur and M. Valero, RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems (**Best Paper Award**), The 2nd ACM International Conference on Performance Engineering (ICPE), March 2011, Karlsruhe, Germany.

4. G. Kestor, L. Dalessandro, A. Cristal, M. L. Scott and O. Unsal, Interchangeable Back Ends for STM Compilers, The 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), June 2011, San Jose, CA.

5. G. Kestor, R. Gioiosa, T. Harris, A. Cristal, O. Unsal, M. Valero and I. Hur, STM2: A Parallel STM for High Performance Simultaneous Multi-Threading Systems, The 20th IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT), October 2011, Galveston Island, TX.

6. G. Kestor, R. Gioiosa, O. Unsal, A. Cristal and M. Valero, Enhancing the Performance of Assisted Execution Runtime Systems., The 17th Architectural Support for Programming Languages and Operating Systems (ASPLOS), Poster Section, March 2012, London, UK.

7. G. Kestor, R. Gioiosa, O. Unsal, A. Cristal and M. Valero, Hardware/Software Techniques for Assisted Execution Runtime Systems, The 2nd Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE), March 2012, London, UK.

8. G. Kestor, R. Gioiosa, O. Unsal, A. Cristal and M. Valero, Enhancing the Performance of Assisted Execution Runtime Systems through Hardware/Software Techniques, 26th ACM/SIGARCH International Conference on Supercomputing (ICS), June 2012, Venice, Italy.

9. I. Kuru, H. S. Matar, A. Cristal, G. Kestor and O. S. Unsal, PaRV: Parallelizing Runtime Detection and Prevention of Concurrency Errors, The 3th International Conference on Runtime Verification (RV), Sep 2012, Istanbul, Turkey.

Chapter 2

Transactional Memory

In this chapter we first describe the Transactional Memory programming model and then we compare TM to classical lock-based programming model, highlighting pros and cons of each. Finally, we conclude the chapter by reviewing the implementation options of TM systems.

2.1 TM Programming Model

While innovations in process technology increase the number of transistors on a die, the performance gains achieved from more complex cores and larger caches diminish. Therefore, chips with multiple cores have quickly become a de-facto standard. Multi-core systems have the potential for significant performance improvements, but the complexity of parallel programming and the difficulty of writing efficient and correct code limit the effective use of these systems.

New programming models have been proposed to ease the development of parallel applications that perform well on multi-core architectures. Transactional Memory (TM) [46, 66, 72] allows programmers to mark compound statements in parallel programs as `atomic` (in C++, `__transaction`), with the expectation that the underlying run-time implementation will execute such transactions concurrently whenever possible, generally by means of speculation—optimistic but checked execution, with rollback and retry when conflicts arise. The principal goal of TM is to simplify synchronization by raising the level of abstraction, breaking the connection between semantic atomicity and the means by which that atomicity is achieved. Secondarily, TM has the potential

to improve performance, most notably when the practical alternative is coarse-grained locking.

2.1.1 Semantics

Transactions have been used in database systems for a long time [51, 127]. In the database world, a transaction consists of a list of operations on the database explicitly declared by the programmer. These operations can be performed in an arbitrary order and do not take permanent effects until the transaction is committed. If there are conflicts caused by other transactions that are concurrently modifying the same data sets, the transaction may be aborted (rolling back its effects) and restarted. TM programming model is based on the same principles and aims to abstract away the complexity of parallel programming from the programmer.

TM systems provide the following properties, which are also referred as *ACI properties*:

- ***Atomicity:*** A transaction encloses a group of instructions to be executed in an atomic way, which means that transactions either complete these instructions in their entirety or behave as if they had never happened.
- ***Consistency:*** Transactions may execute in unpredictable orders, which may lead to an incorrect program execution. A TM system should schedule transactions logically so that their final effect is equivalent to performing transactions serially.
- ***Isolation:*** The execution of transactions performed on shared data does not affect the result of transactions executed concurrently.

Blundell et al. [15, 106] introduced the terms "weak isolation" and "strong isolation". TM systems with weak isolation guarantee transactional atomicity only among transactions, i.e., accesses to shared memory locations within transactions appear as atomic operations to other transactions. TM systems with support for strong isolation, instead, also guarantee transactional semantics between transactional and non-transactional code, hence normal non-transactional accesses are serialized by the TM with any concurrent transactions. Many Hardware TM implementations naturally provide strong isolation, and there has been substantial progress in developing STMs that support strong isolation [2, 140, 143]. Strong isolation facilitates parallel programming

```
HashTable htA, htB;
void move(int key)
{
  int r;
  mutex_lock();
  //acquire the lock
  r = HashTableRemove(htA,key);
  HashTableInsert (htB ,key,r);
  mutex_unlock();
  //release the lock
}
```

(a) Lock-based programming

```
HashTable htA, htB;
void move(int key)
{
  int r;
  atomic
  { //start the transaction
    r = HashTableRemove(htA,key);
    HashTableInsert(htB,key,r);
  }
  //end the transaction
}
```

(b) TM-based programming

Figure 2.1: Lock-based vs. TM-based Programming

for programmers by shifting the management of transactional and non-transactional memory accesses from the programmer to the system: non-transactional accesses are ordered with transactional accesses in a sequential schedule. However, strong isolation requires extra instrumentation barriers that introduce large runtime overhead, especially on STM designs.

2.1.2 Programming

When using TM, programmers replace locks with new language constructs such as `transaction{A}`: this construct executes the statements included in the block of instructions labeled as `A`, as a single transaction. Figure 2.1 illustrates how the atomic constructs can be used in pseudocode. The code in Figure 2.1a shows a possible lock-based implementation of a program that attempts to remove a key from a hash table and add the key to another hash table. Using locks, the programmer explicitly forces all threads to execute any operation between acquiring and releasing the lock serially. Only one thread at a time is allowed to perform any operation on the hash tables. Writing a parallel program as shown in Figure 2.1a is straightforward since the kind of locking used is a coarse-grained lock (we will see later that the price to pay in order to obtain this simplicity is low speedup and, thus, poor scalability).

In Figure 2.1b shows the same algorithm but this time the program uses the aforementioned atomic statement instead of explicit locking. The function calls to

```
HashTable htA, htB; \\ shared Hash tables
int swap(HashTable ht, int key, int newVal)
{
  atomic
  {
    int r = HashTableRemove(ht,key);
    HashTableInsert (ht ,key,newVal);
  }
  return r;
}
...
atomic
{
  int x = swap(htA, key, y);
  HashTableInsert (htB, key, x);
}
```

Figure 2.2: Nesting: The transaction within swap() is nested within the outer transaction by the program control flow. All modifications performed by swap and the outer transaction are executed as one larger transaction; When the transaction in swap() commits, its changes appear in the outer transaction and not globally visible until the outer transaction commits.

`HashTableRemove()` and `HashTableInsert()` in the transaction should be performed atomically with respect to other threads, as if they were done in a single execution step. Unlike the lock-based implementation, the transactional implementation lets all the threads call the functions concurrently as long as they work on different entries of the `HashTable` but any updates on the hash tables become visible only when the transaction commits.

Moreover, transactions provide better scalability than the equivalent lock-based implementation as long as the data-access patterns allow transactions to execute concurrently. Firstly, transactions let two or more threads read the same variable concurrently while basic locking mechanisms do not. This problem can be solved with special read-/write locks which allow multiple concurrent readers at the same time. However, the cost of this improvement is completely on the programmer, who has to make more effort while implementing the algorithm. Secondly, transactions allow concurrent read and write operations to different variables. This is equivalent to write a lock-based program using fine grained locking and provide concurrent accesses to disjoint variables. Again,

with locks, the burden is completely on the programmers. Moreover, beside being a difficult task, the risk of introducing bugs, such as deadlocks, increases.

Summarizing, transactions enable concurrent read accesses to the same memory location and concurrent read and write accesses to disjoint variables while providing the simplicity of coarse-grained locking and achieving the performance that can be obtained by fine-grained locking. The example in Figure 2.1b illustrates these properties: two threads can read the hash-table buckets concurrently and perform a move operation concurrently from different hash-table buckets.

Besides the basic TM primitives intended to start and stop transactions and to annotate memory accesses as transactional accesses, there are advanced TM primitives provided by some TM systems. Nested transactions (see Figure 2.2) allow programmers to create a transaction inside another transaction. The simplest way to support transactional nesting is the flattening model which encloses all transactions in the outermost transaction. With *flat* nesting a conflict in an inner transaction forces all its ancestors to abort. *Closed* and *open* nesting transaction models try to solve this problem. With the closed nesting model [116], nested transactions commit or abort on exit: if a nested transaction commits, its effects become visible only to its parent transaction. If the transaction aborts, its effects are discarded but the parent transaction stays alive. After that, the aborted nested transaction can be re-executed independently from its parent. Open nesting models [120] have more concurrency as compared to closed nesting models: when an open nested transaction commits, all the other transactions can see its updates immediately and continue their work with the new data earlier, without delaying until the outer transaction commits. This may explore more concurrency when shared resources are simultaneously accessed by several large transactions.

Another advanced TM primitive is the “retry”: this primitive enables waiting on multiple conditions such as the “select” system call described in the POSIX standard. Figure 2.3 demonstrates the use of the retry primitives where a thread attempts to find an available data item from a collection of lists. If all of the lists are empty, then the retry statement is executed, which aborts and restarts the transaction.

```
atomic
{
  for (int i=0; i<NUM_LIST; i++)
  {
    int e;
    if (!list[i].empty())
    {
      e = list[i].get_element();
    }
    return e;
  }
  retry;
}
```

Figure 2.3: Example code for the use of "retry" primitive: If the retry statement is reached, the transaction is aborted and re-executed.

2.2 Implementation Options

The key mechanisms of TM systems are data versioning and conflict detection. TM systems must simultaneously manage multiple versions of data while the transactions are still active. In order to achieve this goal, new data versions created by transactional writes are isolated from the rest of the system by maintaining either an undo-log or a write-buffer. When the transaction commits, the new version becomes globally visible. On the other hand, if the transaction aborts, the old version of data remains to be the visible one.

In order to provide a conflict detection, memory accesses in a transactions must be tracked. If a conflict is detected between two transactions, one of them aborts and either restores the old versions of its data from the undo-log or discards its write-buffer, depending on the approach used for data versioning.

2.2.1 Eager and Lazy Data Versioning

As mentioned before, the goal of data versioning in TM systems is to manage different versions of data in a memory and to perform actions when a transaction commits or aborts. More specifically, the data versioning system updates the memory locations with the new values produced by a transaction atomically when the transaction commits. If the transaction aborts, the data versioning system discards the new version of

data safely.

There are two types of data versioning: *eager* and *lazy* data versioning, which can be summarized as follows:

- **Eager versioning** [115] stores transactional write accesses to memory as a new version as soon as possible, and buffers the old version in an undo-log. If the transaction commits, there is no further action required to make the new versions visible. If the transaction aborts, the eager versioning introduces some delay to restore the old versions of data from the undo-log to the memory.

- **Lazy versioning** [43] writes all the new data versions in a write-buffer until the transaction commit phase starts. If the transaction commits, the new versions are copied from the write-buffer to the memory. If the transaction aborts, no further action on the data structures is required and the write-buffer is simply discarded. Unlike eager versioning, lazy versioning introduces delay on the transaction commit phase, since it needs to update the memory locations with the latest version of the data.

2.2.2 Eager and Lazy Conflict Detection

In order to decide whether a conflict between transactions occurs, TM systems track memory accesses through a read-set and a write-set per transaction. The read-set includes addresses read by the transaction and the write-set contains memory addresses written by the transaction. A conflict happens when two transactions access the same address and one of them is a write operation. In particular, conflict detection relies on comparing the read- and write-set of each transaction with all the other read- and write-sets. A conflict is detected when a variable in the write-set of a transaction is also in any set of other transactions.

There are two types of conflict detections: *eager* and *lazy*.

- Systems with **eager conflict detection** check for conflicts as transactions read and write a memory location. Read and write operations are allowed to complete only when they do not cause any conflicts. Under eager conflict detection, conflicts are detected before the end of the transaction so that they can be handled as soon as possible. Resolving conflicts in an early stage reduces the amount of work lost

by aborting transactions. However, the performance of eager conflict detection system depend on which technique is used to resolve conflicts.

- The other technique is **lazy conflict detection**: this approach assumes that conflicts among transactions are rare, thus this technique delays conflict detections to the end of the transaction. While executing a transaction, all the read and write operations are allowed without performing any control. Before committing, the transaction is validated by comparing the read/write sets against the read/write sets of other transactions. If there are no read-write or write-write conflicts, the transaction commits and all temporary data are stored to memory. Lazy conflict detection does not introduce overhead to each read and write as eager conflict detection does, since it postpones all the checks until the end of the transaction.

TM systems can detect conflicts at various granularities. *Word-level* granularity (the smallest possible granularity) provides the highest accuracy but might introduce excessive overhead to track and compare read- and write-sets. This overhead is reduced as the granularity of the conflict detection strategy increases, though the risk of incurring false conflicts increases too. False conflicts are generally undesirable because they might cause more transactions to abort although no real conflicts have actually occurred, which degrades the performance. *Cache-line-level* granularity provides a good trade-off between the false conflicts and the runtime overhead of conflict detection: this design choice divides the memory addresses into a finite set of strips and each strip is mapped to memory locations by using a hash function. The risk of of false conflicts might still be high, depending on the cache line size. *Object-level* detection is an alternative, intended to be used by object oriented applications. Depending on the size of the object, it may reduce the overhead in terms of time (to compare the read sets and the write sets) and space (to track the read sets and the write sets) needed for conflict detection. With this approach, false conflicts only occur when two transactions perform write operations on two different fields of an object. Comparing the three approaches, the word-level granularity is the one with the highest precision and the highest overhead (in terms of both time and space) while the object-level granularity provides the lowest overhead but has a high false conflict risk. The cache-line-level granularity resides between the other two approaches in terms of overhead and the false conflict rate. Unlike the object-level granularity, the cache-line- and word-level

```
int foo(int arg)
{
  //begin the transaction
  atomic
  {
    a++;
  } //end the transaction
}
```

(a) Implicit Transaction

```
int foo(int arg)
{
  BEGIN_TRANSACTION;
    temp_1 = stmRead(a);
    temp_2 = temp_1 + 1;
    stmWrite(a,temp_2);
  END_TRANSACTION;
}
```

(b) Explicit Transaction

Figure 2.4: Type of Used Transactions in Programming Languages

granularity conflict detection systems are not language-level entities, which results in more programming effort to reduce the number of conflicts.

2.2.3 Software and Hardware

Researchers have proposed several different implementations of transactional memory classified into *Software Transactional Memory* (STM), *Hardware Transactional Memory* (HTM) and *Hybrid Transactional Memory*. Software TM systems [5, 43, 65, 67, 70, 71, 104, 133, 135] implement transactional memory entirely in software. An STM implementation instruments all shared memory reads and writes inside atomic sections by using read and write barriers. The instrumentation can be inserted by a compiler in an implicit way (see Figure 2.4a) [35, 138]. In an explicit way, the programmer uses a set of low-level APIs to manually annotate memory accesses in transactions, as shown in Figure 2.4b [23, 43, 104, 133]. As mentioned before, tracking the shared memory accesses is essential for data versioning and conflict detections.

In STM Systems, each transaction has a transaction descriptor that describes the transaction's state which consists of the read/write set (including transaction records), the undo-log (for eager versioning) or the write-buffer (for lazy versioning). Moreover, the transaction's descriptor might include additional data to handle nested transaction with partial rollback. With eager versioning, the write barrier acquires a lock on the transaction record corresponding to the memory location to be updated, then an old value from this memory location is added to the undo-log and then the memory location content is updated with the new value. With lazy versioning, the new value is stored

to the write-buffer with the write barrier; if the transaction commits, the transaction acquires all lock on the all needed transaction records and store all new values from the write-buffer to the memory. In order to detect conflicts among transactions, conflict detection techniques compare version numbers of transaction records in transactions' read/write sets. If a conflict is detected, STM systems provide effective conflict resolution schemes such as stalling one of the transactions or aborting one transaction selected randomly and re-executing the aborted transaction later.

STM systems also provide flexible transactional semantics such as nested transactions with partial rollback. Moreover, STM implementations can accommodate modern language features (e.g, garbage collection (GC), exception handling) and useful tools (e.g, debugger, performance analyzers). However, in general, they incur performance degradation due to the instrumentation required for transactional memory accesses.

Hardware TM systems [10, 27, 62, 72, 115, 131, 155] do not need code instrumentation in the atomic section to manage data versioning and track conflicts, instead, they use a set of instructions in the Instruction Set Architecture (ISA) to provide a low-level transactional interface. In order to guarantee a good performance, it is crucial to cope with data versioning and conflict detection by using hardware resources. Since there is no need to instrument any code, HTM systems are also able to handle more general cases than STM, such as two versions of the same function called from inside transactions or outside of the transaction.

Caches implement data versioning by storing transactional read and write operations to either an undo-log (for eager versioning) or a write-buffer (for lazy versioning). With eager versioning, before performing a new cache line write, the cache line and its address are added to the undo-log by performing additional cache writes. If the transaction aborts, the undo-log must be restored to a memory. With lazy versioning systems, a cache line write is added to the write-buffer with the `W` tracking bit set, which indicates that there is an ongoing write operation. If the transaction aborts, the write-buffer is flushed without performing any validation. If the transaction commits, the data in the write-buffer becomes visible to other transactions and the TM system resets all the `W` bit for each cache line involved in the transaction.

Cache coherence protocols provide communication between read sets and write sets to detect conflicts. With the eager conflict detection mechanism, when a transaction performs a read or a write access, the processor sends a request to the corresponding

cache line. A conflict is detected if there is any copy of the cache line with the `R` (read) or `W` (write) bit set by any processors. The lazy conflict mechanism uses the same coherence message protocol but send all the requests from the write set when the transaction commits.

Even though hardware TM systems offer superior performance, they exhibit additional challenges. Managing data versioning and tracking conflicts transparently by using cache hierarchies and cache coherency protocols are not trivial. Besides that, a long transaction can lead to *cache overflow* since there is a limited space to store all the information related to read/write sets tracking, write buffering and undo logging. Moreover, their restricted semantics do not support transactional language constructs such as deeply nested transactions with partial rollback, blocking primitives and interrupts.

An alternative approach to blend the performance of HTM with the flexibility of STM is Hybrid Transactional Memory [40, 91, 114]. In Hybrid transactional memory implementations, transactions start in the HTM mode, if the HTM system fails due to an excessive resource requirement, the transactions are roll-backed and restarted in the STM mode. Therefore, they are faster than Software TM systems but slower than Hardware TM systems.

There are hundreds of millions of multi core machines already in the field. We believe that, for the sake of backward compatibility, emerging TM-based programming models will need to be implemented in software on these machines. Moreover, a growing consensus holds that STM will be needed as a "fall-back" mechanism when hardware transactions fail due to buffer space limitations, interrupts, or other transient or deterministic causes [36, 42, 91, 144].

2.2.4 Commonly Used STMs

Throughout the dissertation, we use and compare various STM implementations. In the following we summarize the main design choices, characteristics and trade-offs of these popular STMs.

TL2 is an STM that implements a lazy data versioning [43]. A transaction begins by reading the value t in a global "clock." Ownership records (orecs), found by address hashing, indicate the last time at which one of the corresponding location was modified. If a transaction encounters a location that was written after t, it assumes it is

inconsistent, aborts, and retries. At commit time, the transaction locks the orecs for all locations that need to be modified, checks to make sure that all of the locations it read still have a timestamp earlier than t, increments the global time, stores the new time into all the locked orecs, writes out all the updates, and then unlocks the orecs.

TinySTM implements an eager conflict detection along with an eager versioning system with extendable timestamps [133]. Extendable timestamps avoid false positives in which a transaction is aborted despite having seen a consistent view of memory. If a transaction encounters a location that was written after start time t, it checks to see whether any previously read location has been modified since t. If not, it re-reads the global clock and continues, pretending it started at this new time t' instead of t.

TL2 and TinySTM do not support safe privatization. Both require additional code (and nontrivial overhead, not included in our experiments) for correct execution of programs in which data transitions back and forth between shared and private state [105].

RSTM suite includes a variety of STM algorithms, some of which have several variants. The selection of an STM library can be handled simply by re-compiling the code with a different back end. Among the word-based back ends, TML, LLT, ET, NOrec and Pipeline reflect popular but divergent points in the STM design space. These STMs are briefly described below.

- **TML** is an eager conflict detection, eager versioning system with a single sequence lock [92]. TML allows concurrent read-only transactions with no logging overhead, but only one system-wide writer is allowed. This approach is effective in workloads where reads are the common case. However, using a single sequence lock without logging means that conflict detection is extremely conservative: any writer conflicts with any other concurrent transaction.

- **LLT** is a canonical lazy versioning STM implementation patterned after TL2 [43].

- **ET** starts with the basic LLT infrastructure, adds the ability to operate in both eager conflict detection/eager versioning and eager conflict detection/lazy versioning mode, and adds extendable timestamps as in TinySTM [133].

- **NOrec** [39], like LLT, is a lazy versioning system: it delays the resolution of conflicts until a transaction is ready to commit. It uses a single sequence lock [92], however, rather than ownership records to serialize commit and write-back. A

transaction checks, after each read, to see if any writer has committed since start time t; if so, it performs value-based validation [123] to see if its prior reads, if performed right now, would return the values previously seen; if so, as in ET, it reads a new start time from the global clock and continues. Writers can speculate in parallel, but only one can commit at a time. This serialization ultimately limits scalability, but the simplicity of the system yields surprisingly good performance for up to a few dozen cores. Moreover, NOrec is inherently *privatization safe*.

- **Pipeline** extends ET with lazy conflict detection/lazy versioning and it adds the start time linearization approach proposed by Menon et al. [110] to provide single global lock atomicity (SGLA) in Java. SGLA is a basic, pragmatic semantics, where a program is required to behave "as if" transactions were protected by a single global lock. Although SGLA simplifies the design, implementation and testing of STM systems, the implementation of SGLA semantics reduces the scalability because it requires total ordering among all transactions in the system.

Part II

Comprehensive Evaluation of TM Systems

As researchers work to develop robust, mature STM, it becomes increasingly important to be able to effectively and fairly compare STM designs with benchmarks that are representative of real-world applications. Chapter 3 describes RMS-TM, a comprehensive benchmark suite to evaluate (hardware and software) TM implementations. RMS-TM consists of several applications from the RMS domain that are considered representative of future workloads for multi-core systems. Moreover, researchers should be able to share applications, compilers, and run times among groups, and to be able to modify one layer of the system stack while keeping the others constant, for "apples-to-apples" comparison. To this extent, we developed a software layer that allows researchers to test the same applications with interchangeable STM back ends (Chapter 4).

Chapter 3

RMS-TM Benchmark Suite

3.1 Introduction

Muultiple Software TM (STM) [70, 104, 133, 135] and Hardware TM (HTM) implementations [27, 62, 115, 155] have been proposed in the literature. Although some of implementations have reached maturity level, there are still open research issues, in addition to performance, such as handling nested transactions, I/O operations, system and library calls inside transactions. Moreover, performance comparison of TM-based applications against their equivalent lock-based versions is crucial for the justification of further research in this area as well as for convincing the industry to implement TM systems in commercial products. One major aspect of performing functional and performance evaluation of TM systems is the development of an emerging TM benchmark suite.

We identify six desired properties for a TM benchmark suite: (1) the suite should include both the lock-based and TM-based versions of the same benchmarks, (2) the benchmarks should have high scalability, (3) the benchmarks should represent real-world applications, (4) the benchmarks should encompass a wide range of different TM behaviors, (5) the benchmarks should include open research issues for TM researchers, and (6) the benchmark suite should be useful in evaluating both STM and HTM systems.

Although there are multiple benchmark suites [11, 22, 59, 163, 168] proposed for evaluating TM systems, none of those has all of the above-mentioned properties. For example, the *STAMP* benchmark suite [22] does not include lock-based versions of

its applications, *SPLASH-2* [163] does not provide a wide range of TM characteristics, *Atomic Quake* [168] cannot be used to evaluate HTM systems. Previous work by Hughes et al. [74] also pointed out that existing TM workloads have similar characteristics in terms of transactional behaviors and that there is need of more comprehensive benchmarks. In this chapter, we introduce such a benchmark suite, *RMS-TM* (Recognition, Mining, and Synthesis - Transactional Memory). Apart from having a wide range of transactional and run-time characteristics, RMS-TM presents challenging features such as nested transactions, I/O operations and library calls inside transactions, which are common operations in real applications.

To construct our benchmark suite, we develop a step by step methodology for choosing candidate TM benchmarks from among a set of real-world applications, and we reimplement the selected applications by using the TM programming model. The final result is a new benchmark suite that includes different applications from the Recognition, Mining, and Synthesis (RMS) domain. We use RMS applications because these applications have high relevance to mainstream workloads, and they are proposed as emerging workloads to evaluate future multi- and many-core systems [97].

In this chapter we make the following contributions:

- We introduce a new benchmark suite, RMS-TM, that consists of lock-based and transactified versions of seven applications from *BioBench* [8], *MineBench* [119], and *PARSEC* [13] benchmark suites. RMS-TM has a wide range of transactional and run-time characteristics that qualify it as a new and comprehensive benchmark suite for evaluating both STM/HTM designs. The applications in our benchmark suite feature the following: 1) representative real-world applications, 2) nested transactions [116, 120], 3) large amount of I/O operations [12], system [150] and library calls inside atomic blocks, 4) complex function calls and control flow inside atomic blocks, 5) various mix of long/short transactions with different sizes of read/write sets, 6) low/medium/high contention rates, and 7) high scalability.

- We develop a methodical procedure to construct our benchmark suite from candidate applications. We first divide the application selection process into *static* and *dynamic pre-transactification* phases, and then, in the *transactification* phase, we

transactify the selected applications from their original lock-based parallel implementations. This process ensures that the selected applications satisfy the desired properties for a TM benchmark suite.

- We evaluate our benchmark suite using three different TM implementations (one STM and two HTMs), namely *Intel-STM* [135], *EazyHTM* [156], and *Scalable TCC* [27] and we show that RMS-TM can be used in the evaluation of both STM and HTM systems.

  We find that the RMS-TM applications present varying percentage (1.5%-95.7%) of time spent inside atomic blocks with small and large read (a few bytes to 3 MB) and write (a few bytes to 493 KB) sets, and with low and high contention (0.0%-88.4% abort rates). We also find that our benchmarks have high scalability (Intel STM 4.7×, EazyHTM 6.0×, and ScalableTCC 6.3×, on average, for eight threads).

3.2 The Transactification Process

In this section we describe our methodology for constructing the RMS-TM benchmark suite. To create our benchmark suite, we develop a two-step procedure: (1) we apply static and dynamic pre-transactification to select applications from among a set of candidate benchmarks, and (2) we transactify the selected applications.

We analyze three different benchmark suites: BioBench, MineBench, and PARSEC. The applications in these benchmark suites are from the RMS domain, and they represent future workloads [97]. The BioBench suite consists of bioinformatics applications that are developed using the `Pthread` parallel programming model [21]. The MineBench suite is designed considering data mining categories that are commonly used in industry problems. The applications in this suite are implemented by using `OpenMP` [28]. The PARSEC benchmark suite includes emerging applications that are computationally intensive.

3.2.1 Pre-Transactification Phase

We choose applications from the candidate benchmark suites using TM-specific usefulness criteria, e.g., having nested transactions, irrevocable operations, system and

Application	Domain	Locking Type	Nested Locking	Function Calls	Special Operations in Critical Sections	Barrier Synch.
Hmmsearch	sequence profile searching	coarse-grained	no	yes	I/O, memory management operations, library calls	no
Hmmpfam	sequence profile searching	coarse-grained	no	yes	I/O, memory management operations, library calls	no
Hmmcalibrate	calibrate profile HMMs	coarse-grained	no	yes	memory management operations, library calls	no
Apriori	association rule mining	coarse-grained fine-grained	yes	yes	memory management operations	yes
PLSA	dynamic programming	fine-grained	no	yes	none	no
Rsearch	pattern recog- nition mining	fine-grained	no	yes	memory management operations	no no
ScalParC	classification	coarse-grained fine-grained	no	no	none	yes
UtilityMine	association rule mining	coarse-grained fine-grained	yes	yes	memory management operations	yes
Bodytrack	computer vision	fine-grained	no	yes	library calls	yes
Fluidanimate	fluid simulation	fine-grained	no	no	none	yes
Freqmine	frequent item set mining	fine-grained	no	no	memory management operations	no no

Table 3.1: Applications that pass the Static Pre-Transactification step.

library calls inside atomic blocks, etc. To make an effective and comprehensive analysis, we divide the pre-transactification phase into two sub-phases: static and dynamic. In the static phase, we analyze source codes of the applications; in the dynamic phase, we execute and profile the candidate applications to calculate the amount of time they spend inside critical sections and to analyze their scalability.

3.2.1.1 Static Pre-Transactification

We use five criteria in the static pre-transactification phase: (1) synchronization constructs used between lock blocks, (2) type of locking granularity, (3) nested locking, (4) function calls between acquiring and releasing locks, and (5) special operations inside critical sections, e.g., I/O operations, library and system calls.

Table 3.1 shows the characteristics of the applications selected in the static pre-transactification phase. We select *Hmmpfam*, *Hmmsearch*, and *Hmmcalibrate*, because they exhibit a large amount of I/O operations, system and library calls, and relatively complex function calls inside critical sections. *Hmmpfam* and *Hmmsearch* also present a large number of instructions in coarse-grained critical sections. Applications that have a coarse-grained locking structure are promising candidates, because they spend a significant amount of time waiting to acquire a lock; minimizing this synchronization time is an important topic for TM research.

Application	Number of Threads			
	1	**2**	**4**	**8**
Hmmsearch	0.3	0.4	0.4	0.5
Hmmpfam	11.1	12.0	14.2	20.6
Hmmcalibrate	3.9	4.2	4.8	5.6
Apriori-100	1.1	1.6	2.8	5.6
Apriori-1000-20	0.1	0.2	0.4	0.7
Apriori-2000-20	0.1	0.1	0.2	0.4
PLSA	0.0	0.0	0.0	0.0
Rsearch	0.0	0.0	0.0	0.0
ScalParC-A64-D125	0.0	0.2	1.0	1.9
ScalParC-A64-D250	0.0	0.1	0.6	0.8
ScalParC-A64-500	0.0	0.1	0.4	0.7
UtilityMine-1000-10-1	53.9	52.8	56.8	56.3
UtilityMine-1000-10-20	70.1	66.0	70.0	69.5
UtilityMine-2000-20-1	69.8	65.6	69.5	65.7
Fluidanimate	0.0	5.5	9.6	15.2
Freqmine	0.0	0.0	0.0	0.0
BodyTrack	0.1	0.2	0.3	0.2

Table 3.2: Percentage of time spent inside critical sections with respect to total parallel time for the lock-based applications. The data sets used are appended to the application name.

ScalParC, *Apriori*, and *UtilityMine* include both fine- and coarse-grained locking, providing different types and sizes of transactions. In addition, they use synchronization constructs between atomic blocks. We expect placement of synchronization constructs between lock blocks to create interesting TM characteristics, e.g., a high abort rate even when an application does not spend much time inside transactions. In fact, immediately after a barrier, all threads will attempt to enter their atomic blocks at the same time, but only one will commit successfully.

PLSA, *Rsearch*, *BodyTrack*, *Fluidanimate*, and *Freqmine* pass the static pre-transactification phase as well as. Since these applications have function calls inside critical sections, it is difficult to statically determine the length of the transactions and their read/write sets. In addition, some of these applications have memory management operations inside critical sections, such as `malloc()` or `free()`, that are challenging for some TM design.

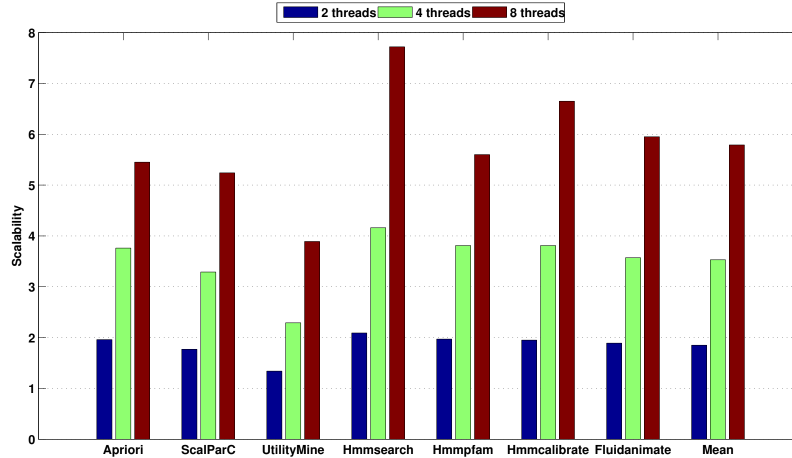


Figure 3.1: Scalability of the lock-based applications, with the largest data sets, normalized to single-threaded lock execution time.

3.2.1.2 Dynamic Pre-Transactification

In the dynamic pre-transactification phase, we use percentage of time spent inside critical sections and scalability as the evaluation criteria.

Table 3.2 shows that *PLSA*, *Rsearch*, *Freqmine*, and *BodyTrack* spend a very small percentage of their execution time inside critical sections. These applications cannot stress the underlying TM systems due to their short transaction lengths, low transaction frequencies, and small read/write sets; therefore, we filter out these applications. Even though *ScalParC* and *Apriori* spend a short amount of time inside critical sections, we maintained these applications in the benchmark suite because they have several marked atomic blocks and they use synchronization constructs, e.g., barriers between consecutive atomic blocks. *Apriori* and *UtilityMine* have a high level (up to nine) of nested locking, which makes them important candidates for evaluating TM systems with support for arbitrary levels of nested transactions. From the *Hmmer* package, we select *Hmmsearch*, *Hmmpfam*, and *Hmmcalibrate*. Although *Hmmsearch* spends a short time inside critical sections, it is a crucial benchmark for TM research because it has I/O operations and library and system calls inside critical sections.

Figure 3.1 shows the scalability of lock-based applications that we consider as promising candidates for TM research. Notice that all the benchmarks have a sub-

linear speedup but they scale well except when we use eight threads in parallel, i.e., all the available processors in our experimental setup.

3.2.2 Transactification Phase

We transactify the selected applications starting from their equivalent lock-based versions by replacing locks with transactions. To maintain the original semantics, we keep the size of the atomic blocks as in the lock-based versions.

The transactification process is not straightforward because each application has a different parallelization strategy. Moreover, each TM system poses specific challenges, e.g., calls to pre-compiled library functions and I/O operations and system calls inside transactions. We now describe the details of these challenges and our solutions for three TM systems, namely Intel STM [135], EazyHTM [156], and ScalableTCC [27].

3.2.2.1 STM Implementation

Intel STM [135] consists of a C/C++ compiler and a STM Runtime Library. The compiler instruments all shared memory reads and writes inside transactions by using read and write barriers. The flattening model is used to support nested transactions, and weak isolation between transactional and non-transactional code is provided. Transactions can be executed in optimistic or pessimistic mode. In both cases, the transactional writes update the data in-place with strict two-phase locking, while the transactional reads are executed optimistically or pessimistically. Serial execution mode is also provided to support transactions that contain irrevocable operations.

Intel STM compiler provides simple language extensions to develop TM applications. The functions inside atomic blocks should be marked as either `tm_callable`[1] or `tm_pure`[2]. Otherwise, if an unannotated function is called inside atomic blocks, the compiler generates code that triggers serial execution unless it knows that the called function does not require instrumentation. The applications that we examine often allocate objects through the `new` operator and/or they call external functions inside

[1] The compiler generates a clone function annotated as `tm_callable` and translates each memory `read` and `write` to a TM read barrier function and a TM write barrier function.

[2] The programmer guarantees that a function marked as `tm_pure` does not access shared variables when it is called from inside a transaction.

atomic blocks. The version of the compiler that we use[1] does not mark the `new` operator as `tm_callable` implicitly although the object constructor is marked. This causes transactions to run irrevocably. To deal with this challenge we overload the `new` operator and we mark it as `tm_callable`. Another challenge is associated with function calls of precompiled libraries inside transactions. To avoid executing these transactions in serial mode, we reimplement some glibc string functions, such as `strcmp`, `strstr`, `strlen`, and we mark them as `tm_callable`.

3.2.2.2 HTM Implementations

EazyHTM [156] and ScalableTCC [27] are HTM proposals that provide scalable performance. Both TM systems are directory-based and implement lazy data versioning. The key feature of EazyHTM is separating conflict detection and conflict resolution. Conflicts are detected while transactions run, but they are resolved at commit time allowing truly parallel commits. On the other hand, ScalableTCC detects conflicts optimistically when transactions are ready to commit. ScalableTCC implements a continuous use of transactions within parallel programs providing non-blocking execution and improved fault-isolation.

The main challenges that we faced while porting RMS-TM applications to EazyHTM and ScalableTCC, are dynamic memory management and I/O operations inside transactions. Most of our applications dynamically allocate memory using `malloc` and `realloc`. To overcome this issue, we use a user mode memory manager that allocates chunks of memory for each thread when the applications start [22]. When a thread requires new memory, the user mode manager takes this memory from its pre-allocated pool and assigns it to the thread without calling the `malloc` system call.

In addition, *Hmmpfam* and *Hmmsearch* perform many I/O operations inside critical sections. The replacement of the locks protecting these critical sections with transactions is not straightforward because rollback can happen at any time during the execution of a transaction, and the transaction can restart at any arbitrary point of its execution. Most current TM systems cannot safely perform I/O or system calls inside transactions. For these operations, we use the library developed by Perfumo et al. [129], which enables the use of I/O operations inside transactions. To provide a fair

[1] Intel C++ STM Compiler Prototype Edition 3.0

comparison, we also modify the lock-based versions of the applications to make them use the same library.

3.3 RMS-TM Overview

We used our pre-transactification process to select applications from the RMS domain, and we transactified those applications to construct the RMS-TM benchmark suite. In this section, we provide the descriptions of the applications in the benchmark suite: *Hmmsearch*, *Hmmpfam*, and *Hmmcalibrate* from BioBench, *Apriori*, *ScalParC*, and *Utility-Mine* from MineBench, and *Fluidanimate* from PARSEC.

TM-Hmmsearch reads a Hidden Markov model (HMM) and searches a sequence database for significantly similar sequence matches. In the transactional version, the threads read the next sequence from an input list of sequences in parallel, and they use transactions to protect the accesses to the input list of sequences. Moreover, the threads share two score lists ranked by per-sequence scores and per-domain scores and a histogram of the whole sequence stores. Transactions are used to protect update operations on these data structures.

TM-Hmmpfam searches a query sequence against a profile HMM database. In the transactional version, each thread accesses the shared profile HMMs database and reads the next profile HMM. This application scores the input sequence against the profile HMM and adds a significant hit to the per-sequence and per-domain top hits lists. Transactions protect the shared file pointer of the HMMs database. Update operations on the shared per-sequence and per-domain top hits lists are also enclosed inside transactions.

TM-Hmmcalibrate calibrates a profile HMM using an artificial database of sequences. After reading the profile HMM, this application generates random sequences; it computes a raw score for each sequence against the profile HMM and it adds this score to a histogram. The increment on the shared counter and the generation of the sequence are enclosed in transactions. Another transaction is used to protect the accesses to the histogram of scores.

TM-Apriori [165] is an Association Rule Mining (ARM) algorithm performed on transactional records in a database. This application uses a hash tree to store

candidates. Transactions are used to protect the calculation of support values and the insertion of a candidate item set into the hash tree.

TM-ScalParC [79] is a parallel formulation of a decision tree classification. The decision tree model splits the records in the training set into subsets based on the values of attributes. This process continues until each record entirely consists of examples from one class. During the partitioning phase, different threads try to simultaneously access a shared counter. Transactions protect the accesses to this shared counter.

TM-UtilityMine [99] is another ARM technique. A utility mining model is developed to identify item sets with high utilities. The utility of an item or an item set can be defined as its usefulness. A single common hash tree stores the candidate item sets at each level of search as well as their transaction-weight utilization. Transactions protect the updates of the utility of item sets and insertion of a candidate into the tree.

TM-Fluidanimate [117] is based on spatial partitioning and uses a uniform grid partitioned to cells to reside fluids. The uniform grid is evenly partitioned in subgrids along cell boundaries. We use transactions to enclose the update particles of the cells that lie on subgrid boundaries.

3.4 Evaluation

We evaluate RMS-TM using three different (one STM and two HTMs) TM systems: Intel STM, EazyHTM, and ScalableTCC. We compare the TM-based implementations of the applications to their equivalent lock-based versions and we analyze their transactional behavior, such as read/write set sizes, abort/commit rates, time spent inside atomic blocks, scalability, etc. We also evaluate the STAMP benchmark suite on the same TM systems and we compare and contrast the results with our benchmark suite.

3.4.1 Intel STM Results

In this Section, we present the evaluation of our benchmark suite using the Intel STM system. All results are the averages of five different executions using three different data sets. We perform our experiments on a Dell PE6850 workstation with 4 dual core x64 Intel Xeon processors running at 3.2GHz equipped with 32GB RAM, a 32KB IL1 and a 32KB DL1 private caches per core, a 4MB L2 cache shared by two cores, and a 8MB L3 cache shared by all cores.

Applications	Read Set (bytes)			Write Set (bytes)			Transactions		
	Min	Mean	Max	Min	Mean	Max	#Commits	#Aborts	Abort Rate (%)
TM-Hmmsearch	24	3K	3M	0	296	493K	613,316	7,678	1.2
TM-Hmmpfam	16	7K	2M	0	846	270K	28,333	5,832	17.1
TM-Hmmcalibrate	8	13K	74K	4	5K	30K	10,016	76,219	88.4
TM-Apriori-100	4	424	67K	0	274	45K	14,410	282	1.9
TM-Apriori-1000-20	4	408	132K	0	263	87K	14,431	290	2.0
TM-Apriori-2000-20	4	449	380K	0	289	246K	14,758	464	3.0
TM-ScalParC-A64-D125	8	31	952	1	7	238	52,404	61,072	53.8
TM-ScalParC-A64-D250	8	30	840	1	7	210	75,408	80,691	51.7
TM-ScalParC-A64-D500	8	34	944	1	8	236	117,240	153,872	56.8
TM-UtilityMine-1000-10-1	32	424	28K	4	7	202	43,724,391	292,031	0.7
TM-UtilityMine-1000-10-20	4	646	65K	4	7	1K	197,213,249	1,212,087	0.6
TM-UtilityMine-2000-20-1	4	644	47K	0	7	1K	3,954,033,044	2,181,138	1.0
TM-Fluidanimate	4	8	1K	4	7	12	1,177,944,500	252	0.0

Table 3.3: Basic TM characteristics (with eight threads) of the RMS-TM applications, with Intel STM. The number of bytes read/written transactionally and the number of aborts or commits are generated by the Intel STM runtime library.

3.4.1.1 Transactional Behavior

Table 3.3 presents the basic runtime TM characteristics of the RMS-TM applications, such as the number of bytes read or written transactionally, the number of times a transaction aborts execution due to a conflict, etc. RMS-TM explores several combinations of TM characteristics: medium read/write sets with medium abort rates (*TM-Hmmpfam*), small read/write sets with high abort rates (*TM-ScalParC*), and large read/write sets with high abort rates (*TM-Hmm calibrate*). In addition, the information presented in Table 3.3 show that the RMS-TM applications cover a wide spectrum of contention ranging from 0.0% for *TM-Fluidanimate* to 88.4% for *TM-Hmmcalibrate*. Although *TM-ScalParC* spends most of its execution time outside atomic blocks, it has a high abort rate due to the use of synchronization points between consecutive atomic blocks, which confirms our observation in the static pre-transactification phase.

Table 3.4 presents the percentage of time spent in atomic blocks with respect to total parallel time with 1, 2, 4, and 8 threads for each data set. We observe some overhead introduced by the Intel STM compiler and run-time library because of the extra work required to handle transactions, such as when detecting conflicts. As we can see from Table 3.2 and Table 3.4, the Intel STM runtime introduces different overheads in the transactified versions of the benchmarks. For example, the lock version and TM version of *TM-Hmmpfam* spend 20.6% and 20.7% of their parallel times inside critical sections. On the other hand, *TM-ScalParC-A64-D250* spends 0.8% of its parallel time inside

Application	Number of Threads			
	1	**2**	**4**	**8**
TM-Hmmsearch	1.1	1.1	1.2	1.6
TM-Hmmpfam	11.1	12.0	14.2	20.7
TM-Hmmcalibrate	7.8	8.3	9.3	14.3
TM-Apriori-100	3.4	5.1	9.7	17.2
TM-Apriori-1000-20	0.0	0.2	0.6	1.8
TM-Apriori-2000-20	0.2	0.3	0.7	1.5
TM-ScalParC-A64-D125	0.1	0.5	2.3	11.5
TM-ScalParC-A64-D250	0.1	0.3	1.5	7.4
TM-ScalParC-A64-D500	0.1	0.2	1.0	5.9
TM-UtilityMine-1000-10-1	88.7	91.8	91.5	92.2
TM-UtilityMine-1000-10-20	95.3	96.0	95.4	95.6
TM-UtilityMine-2000-20-1	95.2	95.8	95.4	95.7
TM-Fluidanimate	0.0	18.9	39.3	61.7

Table 3.4: Percentage of time spent inside atomic blocks with respect to total parallel time for RMS-TM applications.

critical sections with the lock implementation and 7.4% with the TM implementation. Between these two extremes of the spectrum, there are intermediate cases. For example, *Hmmsearch* spends 0.5% in the lock-based version and 1.6% in the transactified version. Table 3.4 shows that the benchmarks cover a wide range of cases in terms of time spent inside atomic blocks. This variety is a desirable property for a TM benchmark suite, because it allows researchers to evaluate TM systems using applications that are either very sensitive to TM overheads (*TM-ScalParC-A64-D250*) or those that are not sensitive to the overhead of TM systems (*TM-Hmmpfam*).

3.4.1.2 Performance Analysis

Figure 3.2 shows the scalability of the RMS-TM applications with respect to their single-threaded case. The RMS-TM applications present a scalability similar to their equivalent lock-based versions except *TM-ScalParC*, *TM-UtilityMine*, and *TM-Fluidanimate*. Several factors may influence the scalability of TM applications, but a high abort rate is the most common reason for poor scalability. Table 3.3 shows that *TM-ScalParC* exhibits this characteristic with 56.8% percent abort rate which causes performance degradation especially with eight threads. Although *TM-UtilityMine* has a low abort rate, this benchmark presents a large number of transactions, each one with large read-/write sets. In other words, each rollback operation is expensive (the cost of each rollback depends on the read/write set size) and it affects performance. We also found

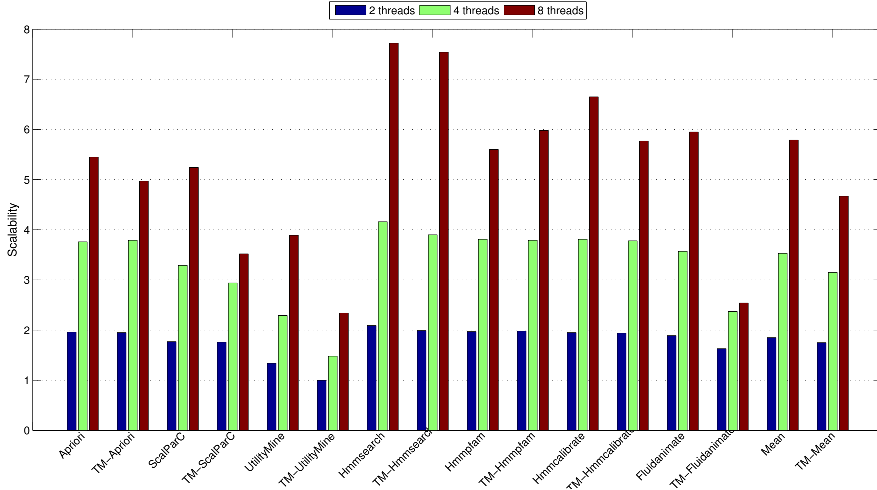
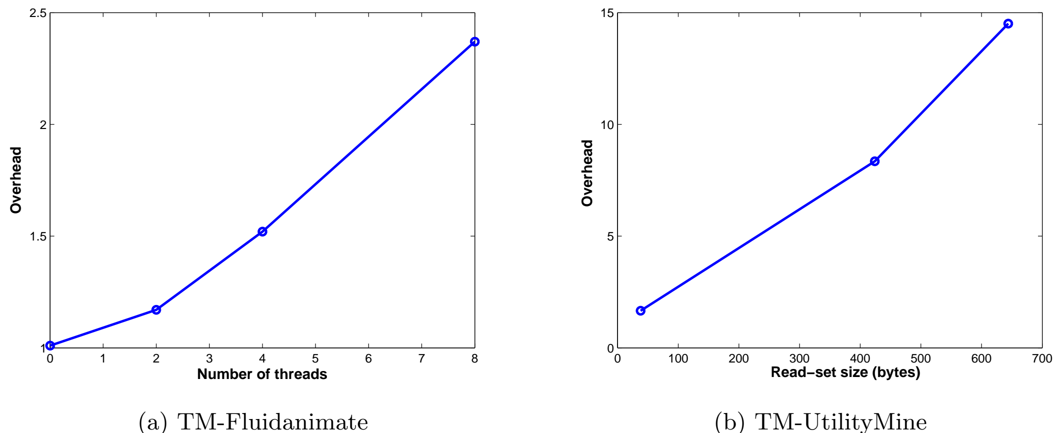


Figure 3.2: Scalability of the lock-based and TM-based applications, with the largest data sets, normalized to single-threaded lock and TM execution time, respectively, with Intel STM.

that *TM-UtilityMine*, with 8 threads, spends 66.0% of its total time inside transactions for rollback operations (wasted work) [128].

We performed a deeper analysis of all the applications using `oprofile` [126] and we examined specific performance counters. We found that the Intel STM run-time system evicts data from the L2 cache while managing the read- and write-sets. This increases the number of L2 cache misses and degrades performance. *TM-UtilityMine* is sensitive to this situation: because of its long transactions with large read sets, more than 90% of the L2 cache misses are caused by the Intel STM library. This extra overhead becomes larger as the sizes of the read- and write-sets increase, therefore, it limits the scalability of the application. Consequently, *TM-UtilityMine* enables TM designers to have better understanding of the runtime overhead of TM systems.

Scalability is also affected by the run-time STM library. Every time a thread attempts to modify a memory location inside a transaction, the STM run-time system scans the read-set of each active transaction to check whether the same memory location was previously read by another thread. The larger the read-set the longer the time required to scan each active transaction and the larger the overhead introduced by the STM run-time system, which limits scalability. On the other side, the larger the num-



(a) TM-Fluidanimate

(b) TM-UtilityMine

Figure 3.3: (a) Runtime overhead of *TM-Fluidanimate* as a function of the number of threads (from 1 to 8) with constant read- and write-set sizes. (b) Run-time overhead of *TM-UtilityMine* increases as the size of the read-set increases.

ber of concurrent active transactions (which is upper bound by the number of threads), the larger the overhead. Figure 3.3a shows the runtime overhead of *TM-Fluidanimate* as function of the number of threads (from 1 to 8) with constant read- and write-set sizes. The run-time overhead linearly increases with respect to the number of threads.[1] In addition, Figure 3.3b demonstrates that the run-time overhead of *TM-UtilityMine* increases with the size of the read-set.[2] Note that applications with high abort rates will interrupt the list traversal sooner because of conflict detection, and they will have a lower performance degradation. Obviously, applications that spend a large part of their execution inside transactions are affected more by the STM run-time overhead.

3.4.2 EazyHTM Results

We evaluate the performance of RMS-TM applications on EazyHTM [156] using a full-system simulator based on the Alpha 21264 architecture. EazyHTM is implemented using the M5 simulator [14] which is modified with a directory memory hierarchy and a core-to-core interconnection network. Table 3.5 presents the main characteristics of the simulated system. We use the largest possible data set in our simulations.

[1]For this application the number of transactions per thread is constant.
[2]The number of threads (eight) is constant in this graph.

Feature	Description
CPU	1-8 Alpha cores, 2 GHz, in-order, 1 IPC
L1	32 KB, 64-byte cache line, 4-way associativity, private per core, writeback, MSI, 2 cycles latency
L2	512 KB, 64-byte cache line, 8-way associativity, private per core, writeback, 8 cycles latency
Main Memory	100 cycles latency
ICN	2D Mesh topology, 10 cycles latency per hop

Table 3.5: Configuration of the simulated system.

Application	Read Set (cache lines)		Write Set (cache lines)		Transactions		
	90 pctile	Max	90 pctile	Max	#Commits	#Aborts	Abort Rate (%)
TM-Hmmsearch	161	975	56	1,368	2,008	362	15.3
TM-Hmmpfam	3,348	10,338	1,400	3,832	308	345	52.9
TM-Hmmcalibrate	51	71	29	37	5,016	376	7.0
TM-Apriori-100	11	40	6	206	11,232	36	0.3
TM-ScalParC-A64-D125	4	4	3	3	50,393	18,979	27.4
TM-UtilityMine-1000-10-1	65	120	1	2	43,724,391	374,050	0.8
TM-Fluidanimate	2	2	1	1	9,347,885	3,131	0.0

Table 3.6: Transactional behavior of the RMS-TM applications with eight threads, with EazyHTM. The sizes of transactional read and write sets are presented as the 90th percentile.

3.4.2.1 Transactional Behavior

Table 3.6 summarizes the transactional characteristics of the RMS-TM applications on EazyHTM. *TM-Hmmpfam* exhibits a high abort rate. This is caused by the large read/write sets that do not fit in the cache. Since EazyHTM does not provide support for unbounded transactions, transactions are eventually aborted and restarted. Benchmarks with high commit rates (*TM-Hmmcalibrate*, *TM-Apriori*, *TM-UtilityMine*, and *TM-Fluidanimate*) and with high abort rates (*TM-Hmmsearch*, *TM-Hmmpfam*, and *TM-ScalParC*) are important candidates to evaluate both lazy and eager data versioning. For example, Hammond et al. [62] and Moore et al. [115] show that high commit/abort rates have large impacts on performance in HTM systems. This happens because eager data versioning relies on the idea that the commit rate is higher than the abort rate, therefore, these systems are designed with a low commit cost. On the other hand, HTM systems with lazy data versioning do not rely on this hypothesis and they usually show that the abort cost is significantly lower than the commit cost. To enable researchers to perform exhaustive studies on TM systems with different

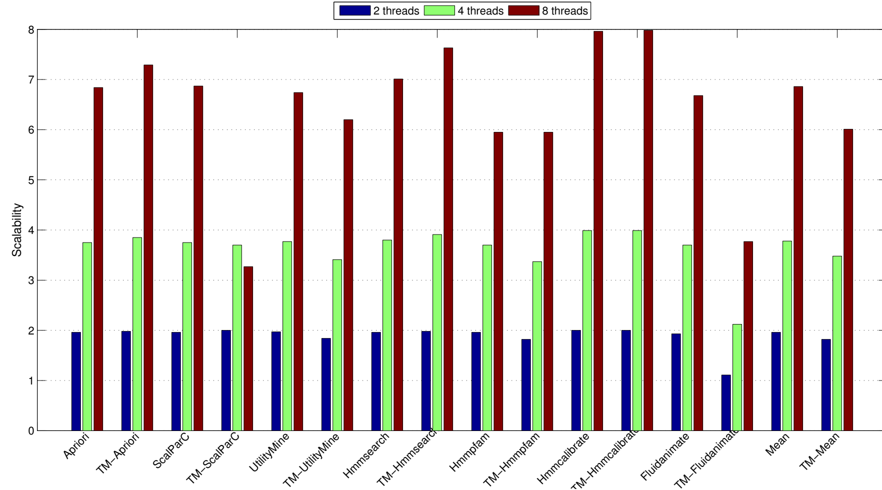


Figure 3.4: Scalability of the lock-based and TM-Based applications normalized to single-threaded lock and TM execution time, respectively, using EazyHTM. The datasets are indicated in Table 3.6.

versioning strategies, RMS-TM provides different combinations of TM behaviors.

3.4.2.2 Performance Analysis

Figures 3.4 shows the scalability of lock- and TM-based RMS-TM applications on EazyHTM. The majority of the TM-based applications exhibit high scalability, comparable to their equivalent lock-based versions. More in details, *TM-Hmmcalibrate* scales linearly, *TM-Hmmsearch* and *TM-Apriori* scale slightly better than their lock-based version, while *TM-UtilityMine* scales slightly worse than its lock-based version. *TM-ScalParC* scales well up to four threads. However, this application scales poorly with eight threads as opposed to its lock-based version. We noticed that the number of directory messages to detect conflicts is constant with two and four threads (where the application shows a reason scalability) but it doubles with 8 threads (the case of poor scalability). *TM-ScalParC* is the only application with such behavior. *TM-Fluidanimate* presents a very high number of directory messages that increases with the number of threads. For all the other applications the number of directory messages is roughly constant regardless of the number of threads. We conclude that EazyHTM's conflict detection mechanism introduces overhead that limits the scalabil-

Application	Read Set (cache lines)		Write Set (cache lines)		Transactions			Wast
	90 pctile	Max	90 pctile	Max	#Commits	#Aborts	Abort Rate (%)	
TM-Hmmsearch	109	945	56	1,369	2,008	204	9.2	0.2
TM-Hmmpfam	3,260	10,342	1,312	3,833	308	219	41.6	8.4
TM-Hmmcalibrate	47	72	26	37	5,016	285	5.3	0.1
TM-Apriori-100	11	19	6	103	14,438	14	0.1	0.6
TM-ScalParC-A64-D125	4	5	3	4	50,352	10,010	16.6	12.9
TM-UtilityMine-1000-10-1	65	120	1	3	43,724,391	436,698	1.0	1.7
TM-Fluidanimate	2	2	1	1	9,347,885	2,207	0.0	0.1

Table 3.7: Transactional behavior of the RMS-TM applications with eight threads, with ScalableTCC. The sizes of transactional read and write sets are presented as the 90th percentile.

ity of *TM-ScalParC* with eight threads and *TM-Fluidanimate* with two, four and eight threads.

3.4.3 ScalableTCC Results

In this section, we present our experimental results for the ScalableTCC HTM system using a full-system simulator based on the Alpha 21264 architecture. Table 3.5 presents the main parameters of the simulated multi-core system that we use for ScalableTCC.

3.4.3.1 Transactional Behavior

Table 3.7 presents the basic TM characteristics of the RMS-TM applications, and it includes data such as the number of commits/aborts and read/write set size in 64-byte cache lines. All transactional characteristics in Table 3.7 show that RMS-TM covers different combinations of TM execution scenarios, such as the sizes of transactional read (2 - 3,260 cache lines) and write (1 - 1,312 cache lines), and abort rates (0.0% to 41.6%). More specifically, *TM-Hmmpfam* has the largest read- and write-sets, 3,260 (203 KB) and 1,312 (82 KB) cache lines, respectively. Moreover, this application presents the highest abort rate (41.6%). Effective contention manager policies can reduce the number of aborted transactions, which implies that *TM-Hmmpfam* can enable TM designers to improve their contention manager proposals. On the other hand, *TM-UtilityMine* and *TM-Fluidanimate* show high commit rates with a large number of committed transactions, which makes them desirable TM benchmarks for evaluating TM systems with lazy data versioning, where the commit cost is high.

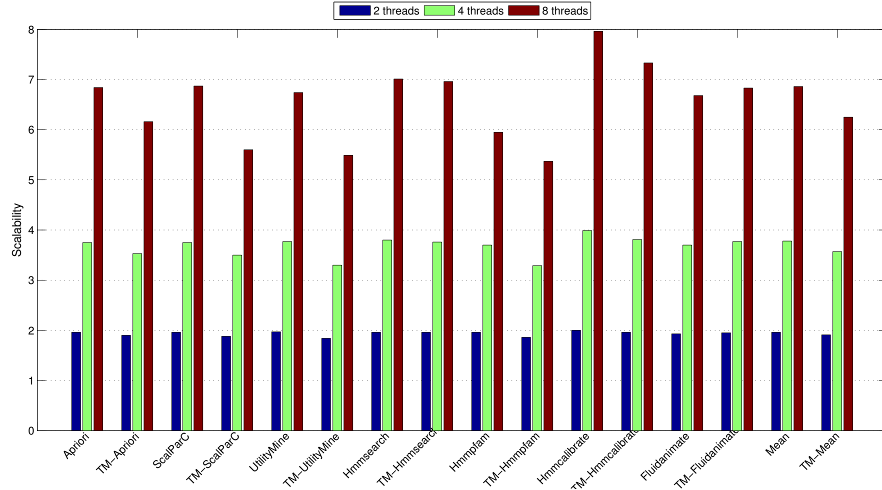
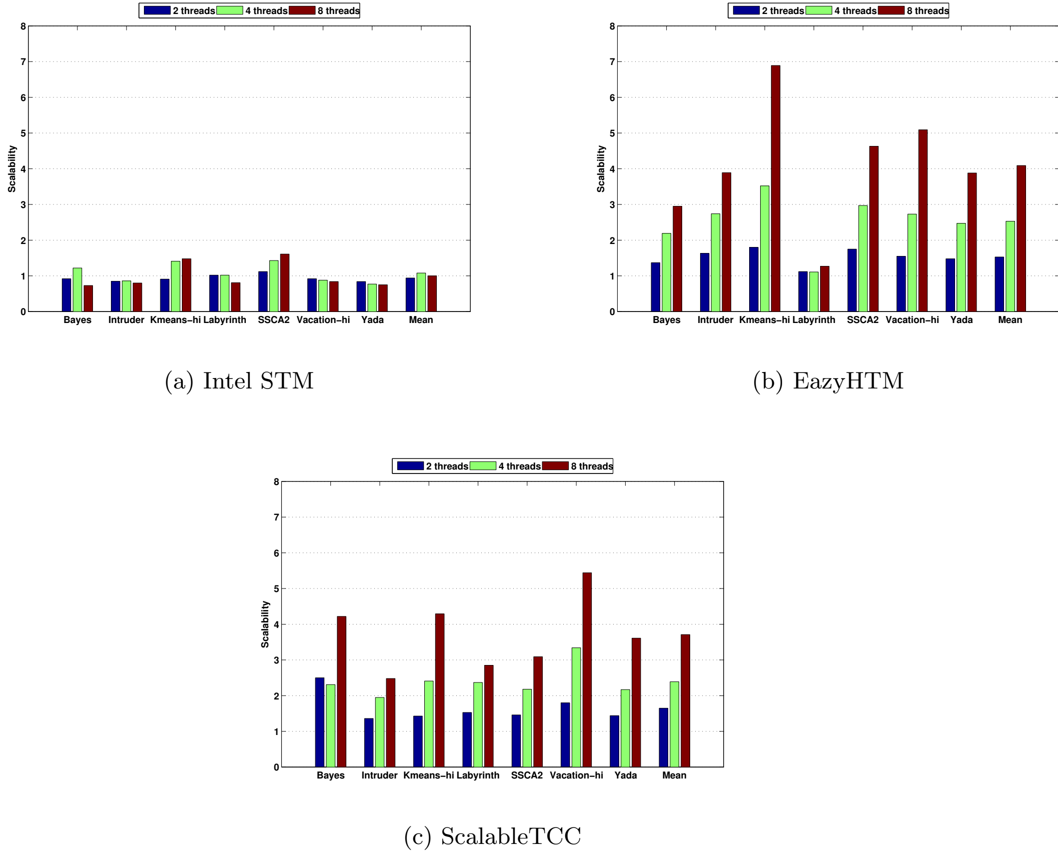


Figure 3.5: Scalability of the lock-based and TM-based applications normalized to single-threaded lock and TM execution time, respectively, with ScalableTCC. The datasets are indicated in Table 3.7.

Figures 3.5 shows the scalability of lock- and TM-based RMS-TM applications on ScalableTCC. Most of the TM-based applications present similar scalability to their equivalent lock-based versions except *TM-ScalParC*, *TM-Apriori*, *TM-UtilityMine* and *TM-Hmmpfam*. As we can see from Table 3.7, *TM-ScalParC* and *TM-Hmmpfam* with eight threads waste 12.9%, and 8.4% of their total execution time, respectively, which limits their scalability. Further analysis showed that rolling back aborted transactions is a large component of the total wasted time for these applications. *TM-Apriori* and *TM-UtilityMine* do not scale as well as their lock-based equivalent with eight threads. For these applications, we observe that they spend relatively large amount of time at synchronization points especially with eight threads, as opposed to the other applications.

3.4.4 Comparison of RMS-TM and STAMP

In this section we compare RMS-TM to STAMP using three different TM systems. Both RMS-TM and STAMP have substantial number of applications with varying abort /commit rates and small/large transactions. RMS-TM also has I/O operations, library calls, memory management operations, pre-compiled library calls inside transactions,



(a) Intel STM

(b) EazyHTM

(c) ScalableTCC

Figure 3.6: Scalability of the STAMP applications normalized to single-threaded TM execution time.

and nested transactions, whereas STAMP only provides memory management operations inside transactions. Hence, we believe that the RMS-TM applications present more realistic use cases of TM. On the other hand, the STAMP benchmarks provide larger read/write sets than RMS-TM. This characteristic can help TM researchers evaluate their TM proposals that support virtualized transactions [32].

We analyze the scalability of RMS-TM and STAMP applications on three different TM systems. Figures 3.6a, 3.6b and 3.6c, show that RMS-TM applications scale well as the number of cores increases on both STM and HTMs (Intel STM 4.7×, EazyHTM 6.0×, and ScalableTCC 6.3×, on average, with eight threads). However, some STAMP applications on the STM implementation, Figure 3.6a, show no scalability regardless of

the number of threads, whereas they have a reasonable scalability on HTMs (EazyHTM 4.1× and ScalableTCC 3.7×, on average, with eight threads).

Unlike STAMP, RMS-TM provides both lock-based and transactified implementations to better understand drawbacks of TM proposals through direct performance comparison. For example, as presented in Section 3.4.1.2, performance and scalability analysis between *TM-UtilityMine* and its equivalent lock-based implementation provide important insights into the STM system. With this information we understand that *TM-UtilityMine's* poor scalability is caused by STM run-time overhead rather than the algorithm. Finally, RMS-TM consists of applications written in C/C++ programming languages using different parallel programming models such as `OpenMP` and `Pthread`. On the other hand, STAMP applications are implemented in C with `pthread`.

3.5 Related Work

In this section, we briefly review some of the previously proposed TM benchmarks to highlight their advantages and disadvantages in evaluating TM systems. We categorize TM benchmark suites into micro-benchmarks (*TM micro-benchmarks* [50] and *The Haskell STM Benchmark Suite* [128]), parallel benchmarks (*SPLASH-2* [163]), and other benchmarks with more complex transactional characteristics (*STMBench7* [59], *Lee-TM* [11], *WormBench* [167], *STAMP* [22], *Atomic Quake* [168]. and *QuakeTM* [55]).

TM micro-benchmarks contain single data structures, such as hash tables, linked lists, B-trees, etc. These benchmarks are useful for providing basic-level insights into TM designs, but they do not exhibit different TM characteristics, and they are not representative of realistic workloads.

The Haskell STM Benchmark Suite consists of ten applications that are implemented with Haskell, which features TM as a first-class language feature. Most of the applications in this benchmark suite are micro-benchmarks.

SPLASH-2 contains eight parallel applications and four computational kernels. This benchmark suite focuses on applications that utilize little synchronization between threads, and it does not provide various sizes of critical sections or different conflict rates. Therefore, this benchmark suite is not fully capable of evaluating the underlying TM system and discovering interesting transactional behaviors.

STMBench7 presents an application to analyze STM systems. This benchmark provides a coarse-grained and medium-grained locking implementation that can be compared to its equivalent transactified version. The benchmark performs complex and dynamic operations on a large data structure, so it has only relatively long transactions.

Lee-TM benchmarks feature long and realistic workloads that consist of sequential as well as coarse- and medium-grained lock-based, transactional, and optimized transactional implementations. This benchmark suite is useful for comparing different lock and transactional implementations; however, it only features different implementations of the same algorithm.

WormBench is a highly configurable transactional application. This synthetic application is useful mostly to mimic existing TM applications rather than discovering unknown usage patterns of emerging transactional applications.

STAMP is a benchmark suite that consists of eight applications with 30 different sets of configurations. The input data for the applications present a wide range of runtime transactional characteristics, e.g., varying transaction lengths, read/write set sizes, and degree of contention. This benchmark suite provides sequential and transactional versions of the applications, but it does not provide their lock-based versions; thus, TM researchers cannot compare TM-based and the equivalent lock-based implementations.

QuakeTM and *Atomic Quake* are rich and complex transactional memory applications. QuakeTM is parallelized from the sequential version of Quake game server using TM, while Atomic Quake is derived from the parallel lock-based version of the server. These benchmarks exhibit irregular parallelism, have I/O and system calls, error handling, and instances of privatization. In addition, inside transactions, there are function calls, memory management, and nested transactions. However, these benchmarks can only be used for evaluating STM systems due to their size and complexity.

In comparison, RMS-TM includes lock-based and TM-based implementations of seven real-world applications that have a wide range of TM characteristics in terms of transaction lengths, read/write set sizes, and contention. This benchmark suite is suitable for evaluating both STM and HTM systems. In addition, unlike most other TM benchmarks, RMS-TM presents many desirable properties, such as nested transactions, I/O operations, and system calls inside transactions.

3.6 Conclusions

We introduced a new TM benchmark suite, RMS-TM, that consists of multi-core workloads from the Recognition, Mining, and Synthesis domain. We developed a general methodology to determine applications that are suitable for analyzing TM implementations, and we transactified the selected applications. Therefore, RMS-TM includes both locked-based and transactified versions of the same applications. We evaluated RMS-TM using one STM and two HTM implementations, and we presented the detailed analysis of our experimental results. We found that the applications in our benchmark suite have high scalability, and they feature a wide range of transactional characteristics. RMS-TM is publicly available in the hopes of helping researchers to design and evaluate their TM systems [134].

Chapter 4

Interchangeable Back Ends for STM Compilers

4.1 Introduction

As researchers work to develop robust, mature STM, it becomes increasingly important to be able to share applications, compilers, and runtimes among groups, and to be able to modify one layer of the system stack while keeping the others constant, for "apples-to-apples" comparison.

Until recently, most STM systems were implemented as user-level libraries: the programmer was expected to manually instrument not only transaction boundaries, but also individual loads and stores within transactions. This library-based approach was adequate for early experiments with microbenchmarks, but it becomes increasingly tedious and error prone for larger applications [37]. The use of different library interfaces in different research groups has also made it difficult to share applications across groups, or to make reliable performance comparisons: experiments with different versions of the application source code inevitably raise questions of fairness and confidence.

A recent draft standard for transactions in C++ [6], and the release of compilers conforming to that standard, promises to significantly ease the construction of large transactional programs, and reduce the problem of source-level incompatibility among groups. Compilers also improve the interoperability of hardware and software TM, by automatically generating the instrumented loads and stores that are required by the latter but not the former. In the software case, the fact that calls to the back-end

system are being generated by a compiler rather than a human programmer means that the back end can provide a wide, performance-oriented ABI instead of a narrow convenience-oriented API.

Unfortunately, much of the work on STM systems over the past 7 years remains incompatible with recent compilers because of interface issues. Indeed, the four publicly available C++ TM compilers support remarkably little back-end diversity. Oracle's compiler, which generates code only for the SPARC, employs the SkySTM back end [95]; Intel's compiler, for the x86, employs a modified version of the STM presented in Ni et al. [121]; and the Dresden and GNU compilers, also for the x86, employ TinySTM [52]. At the same time, the three x86 compilers and their two back ends employ (for the most part) a common ABI designed by Intel [77], which raises the prospect of interoperability.

The RSTM suite [147] comprises the widest diversity of STM algorithms currently available (13 in the version 5 release). In the interest of wider experimentation, we have adapted the "word-based" algorithms to the Intel ABI, allowing them to be used with any conforming compiler. To minimize per-algorithm effort, we introduce a "shim" layer that embodies most of the adaptation. As of this writing, we have successfully connected the Intel C++ TM compiler to three RSTM back ends: LLT (lazy detection, lazy versioning, with timestamps), which resembles TL2 [43]; ET (extendable timestamps), which resembles TinySTM; and Precise (a.k.a. NOrec [39]), which provides unusually strong privatization semantics, and works particularly well as the software half of a hybrid TM system [36].

Our compiler-ready back ends allow us, for the first time, to run large applications on top of RSTM without hand-instrumenting loads and stores. As a first installment toward "apples-to-apples" comparison, we present performance results in Section 4.4 for both RSTM and the Intel back end on several applications from the RMS-TM benchmark suite [86]. We also present results for a selection of microbenchmarks and for applications from the STAMP suite [22]. For STAMP we consider both the original code, which uses hand instrumentation of (only) "important" loads and stores, and new versions written to the C++ TM standard. One new version lets the compiler instrument everything inside transactions; another uses Intel's `__transaction [[waiver]]` extension to disable instrumentation of many "unimportant" loads and stores. Our results suggest that the scalability of STAMP depends critically on minimizing instrumentation.

4.2 Design and Implementation

4.2.1 Draft Specification for TM in C++

The draft standard for C++ TM [6], written jointly by representatives of Intel, Oracle, and IBM, defines language extensions for TM applications. In particular, the `__transaction{}` construct brackets sequences of statements to be executed "all at once."

A transaction can be declared as either `atomic` (the default) or `relaxed`. Atomic transactions are restricted to perform only *safe* operations—loosely, those that a compiler and runtime are sure to be able to execute speculatively, and roll back on abort. In a data-race-free program, an atomic transaction never appears to interleave with execution in other threads or with behavior in the outside world.

Relaxed transactions are allowed to perform *unsafe* operations. They may or may not be executed speculatively. Operations inside a relaxed transaction are isolated from other transactions, but may, if unsafe, appear to interleave with (nontransactional) execution in other threads or with the outside world—even if the overall program is data-race free.

Functions called in an atomic transaction must be declared with the `transaction_safe` attribute, and cannot themselves contain unsafe operations, or calls to unsafe functions. Functions called in a relaxed transaction may be declared with the `transaction_callable` attribute, to increase the likelihood that the compiler will be able to execute the transaction speculatively. A `transaction_callable` function, like a relaxed transaction, is permitted to perform unsafe operations. The compiler can be expected to generate two clones of a `transaction_safe` or `transaction_callable` function—one for use outside transactions, one (with instrumented loads and stores) for use inside. The C++ draft standard calls for transactional function pointers to be statically typed with the same `transaction_safe` or `transaction_callable` attributes as the functions being assigned into them.

Some unsafe operations are said to be *irrevocable*, meaning that they cannot be rolled back. Examples include I/O and writes to `atomic` variables. If a relaxed transaction performs an irrevocable action, the STM implementation can be expected to preclude concurrent execution of certain other transactions [152]. Note that not all un-

safe operations are necessarily irrevocable. For example, a read of a `volatile` variable is an unsafe operation but it will probably not be irrevocable.

The Intel compiler, which we use for our experiments, implements certain extensions to the C++ TM standard. For example, a function can be declared with the `transaction_pure` attribute, meaning that the programmer guarantees it to be idempotent, and thus safe to execute—even within an atomic transaction—without instrumentation on its loads and stores. Finally, the `__transaction [[waiver]] {}` construct can be used to bracket a sequence of statements inside a transaction that should not be rolled back on abort. Waivered code is essentially unstructured open nesting; example use cases include debugging, statistics gathering, and semantically neutral operations like tree rebalancing.

The current version of Intel's compiler does not implement transactionally typed function pointers. It supports the transactional use of function pointers by dynamically detecting if the indirect call target has a transactional clone, calling it if it does, and switching to *serial irrevocable* mode to perform the indirect call nontransactionally if it doesn't. This has two side effects: indirect calls through function pointers are only valid in `relaxed` transactions as they might require *serial irrevocable* execution, and incorrectly annotated source may lead to poor performance due to transactions silently switching to *serial irrevocable* mode.

4.2.2 Intel ABI Overview

Figure 4.1a shows a simple program fragment using the C++ TM API. The Intel compiler automatically generates an equivalent version instrumented for the Intel ABI (Figure 4.1b). Implementations of the functions in the ABI are provided by the underlying STM library. This subsection describes the instrumentation performed by the Intel compiler; the following section details how we link the instrumented code to the RSTM back ends.

The code in Figure 4.1a executes a transactional read of variable `a`, increases its value by 5, and writes back the result (line 5). In Figure 4.1b, the thread performing the transaction allocates a *transaction descriptor* by calling `_ITM_getTransaction` (line 3). The `beginTransaction` function (line 4) takes several parameters: the transaction descriptor, a set of bit values encoding information about the transaction's properties, and the source location where the atomic block begins. Given these, `beginTransaction`

```
int a;
int foo()
{
  __transaction{
    a = a + 5;
  }
}
```

(a) C++ TM standard application programming interface (API)

```
int foo()
{
  _ITM_transaction * td =
  _ITM_getTransaction();
  int doWhat =
    beginTransaction(td, prop, str_loc);
    /* a = a + 5; */
  int a_tmp = _ITM_RfWU4(td, &a);
  a_tmp = a_tmp + 5;
  _ITM_WaWU4(td, &a, a_tmp);
  _ITM_commitTransaction(td, &outer_commit);
}
```

(b) Intel TM application binary interface (ABI)

Figure 4.1: Automatic read/write instrumentation of a simple TM program

saves the machine state (callee-saves registers, stack pointer) and starts the transaction. If the transaction aborts internally, execution will resume with a second return from `beginTransaction`–it effectively has `setjmp` semantics in this way.

At line 6, the compiler knows that the read of `a` will be followed by a write. It therefore instruments the access with a call to `_ITM_RfWU4`—Read for Write, 4 bytes. In an eager-acquire STM this routine could pre-acquire a write lock on `a`, avoiding the need to promote a read lock later, and return the value of `a`, which the compiler saves in temporary variable `a_tmp`.

The next write operation is instrumented with `_ITM_WaWU4`—Write after Write, 4 bytes, which can avoid the complexities of lock promotion. The `_ITM_WaWU4` (line 8) updates `a` with its new value (`a_tmp`). The last call in the generated code (`_ITM_commitTransaction`) attempts to commit the transaction. If the function detects that the transaction has conflicts, then the transaction will abort and perform a `longjmp` back to `beginTransaction`. If no conflicts are detected, the transaction commits and execution continues with whatever lies after line 10.

4.2.3 Design Details

Several technical challenges made the adaptation of RSTM to Intel ABI an interesting and nontrivial task. As noted in Section 4.1, the principal design decision was to introduce a "shim" library that maps the ABI function calls generated by the compiler (sometimes with a bit of "glue" code) to the function signatures provided by (one of) the RSTM back ends. This strategy allows most of the adaptation work to be done once rather than once per back end. The main disadvantage of the shim approach is potentially extra overhead. Fortunately, most of the back end routines in RSTM were intended to be inlined into manually instrumented source. We inline them into the shim instead, allowing us to incur only one function call, rather than two, at each instrumentation point.

Subword accesses. The existing RSTM back ends were designed to support only 4-byte loads and stores, but the Intel ABI requires 1-, 2-, 4-, 8-, 12-, 16-, 24-, and 32-byte accesses as well. Multiword accesses are easily implemented (if slightly inefficiently) as sequences of word accesses. Subword accesses, however, raise the possibility of false sharing. If x and y occupy opposite halves of the same word, for example, then a transaction that modifies x may force the abort of a transaction that reads y, even though no conflict has actually occurred. Worse, if nontransactional code modifies y during the execution of a transaction that modifies x, commit-time write-back of the word containing x may overwrite the modification of y, leading to incorrect behavior—even though the program is data-race free.

Perhaps the simplest solution would be to maintain read and write logs at byte granularity, but this would quadruple the cost of instrumentation for common-case word-sized accesses. A second alternative might be to maintain separate logs for word, halfword, and byte level access, but this leads to significant complexity when a transaction accesses the same word at multiple granularities. We ultimately chose to add a bit mask to each entry in the read and write logs, to identify which part(s) of the word have been accessed. Appropriate bits are or-ed into the mask on each access. During write-back, only modified bytes are updated. During value-based validation (as in NOrec), only accessed bytes are compared.

For orec-based conflict detection (as in LLT and ET), we see no easy way to keep track of subword updates. Per-byte timestamps would again quadruple the cost of

common operations, and bit mask schemes suffer from the fact that different words mapping to the same orec may have different update patterns, and different bytes may be updated at different times. For the sake of simplicity and modest overhead, we have chosen to maintain orec-based conflict detection at the word level only. This can lead to unnecessary aborts, but not to incorrect behavior.

Two small optimizations streamline the code path for load and store instrumentation. First, a "fast path" always checks for full-word granularity, since that is the common case. Second, to simplify masking, bitmaps are full-word width, with 8 identical bits in every byte.

Inevitability (irrevocability). The Intel ABI defines a function (`changeTransactionMode`) that can be used to make completion of a transaction inevitable prior to I/O, calls to uninstrumented functions, or other irreversible operations. The RSTM back ends currently support inevitability only when requested prior to performing any loads or stores. To support the Intel ABI routine, we arrange to abort a transaction that has already performed memory accesses, and restart it in inevitable mode.

Missing functionality. Support for some of the Intel ABI routines was missing entirely in RSTM and had to be added to the shim. The `addUserUndoAction` and `addUserCommitAction` routines allow user code to register functions to be called when a transaction rolls back or commits. In the absence of explicit guidance in the ABI, we arrange to call these functions in the order in which they were registered. The `registerThrownObject` routine allows user code to register exception objects. Updates to such objects are not rolled back on abort, and for redo-log implementations buffered writes to such objects must be performed during aborts.

4.3 Experimental Setup

In the Section 4.4 we use our Intel/RSTM shim to (1) explore the overhead of automatic (as opposed to manual) read and write instrumentation, and (2) compare the performance of the default Intel back end to three of the RSTM alternatives. In our experiments we employ three RSTM microbenchmarks (HashTable, DoubleList, and RBTree) and selected applications from the STAMP [22] and RMS-TM benchmark [86] suites.

The STAMP suite comprises eight applications with 30 configuration sets. The applications are drawn from bioinformatics, engineering, computer graphics, and machine learning. They vary significantly in transaction lengths, read- and write-set sizes, and degree of contention. All were written with explicit calls to a transactional library API. They needed to be modified by hand to employ the C++ TM standard API instead. In the time available we were able to complete three of the eight applications: Kmeans, SSCA2, and Vacation. Kmeans and SSCA2 were straightforward: their transactions are relatively simple, with no nested subroutine calls, transactional `libc` library calls, or unsafe operations. Vacation was more of a challenge (as would be the five remaining applications). We annotated functions called from within transactions in Vacation as either `transaction_safe` or `transaction_callable`, depending on whether they include unsafe operations. We then defined transactions as `atomic` or `relaxed` accordingly.

STAMP implements generic data structures using function pointers. A set of objects of opaque type, for example, is represented with a list of `void*` and a pointer to a function that can be used to test for object equality. STAMP's initial implementation uses pointers to uninstrumented functions in such contexts: the original developers determined that the lack of instrumentation would not compromise program correctness. As described in Section 4.2.1, the Intel compiler currently generates code that will silently switch to *serial irrevocable* mode when it encounters such pointers. To mimic the behavior of the original STAMP application, we can use Intel's `__transaction [[waiver]]` extension, which allows us to call through these pointers nontransactionally. Alternatively, we can declare the target functions as `transaction_safe` and call them transactionally, without the waiver. This leads to significant overhead, however, because the functions are called frequently during core data structure traversals, and the compiler must now use instrumented versions of the code. For completeness we test both "with waiver" and "without waiver" versions of Vacation.

The RMS-TM suite comprises seven applications from the Recognition, Mining and Synthesis (RMS) domain. As in STAMP, transactions vary greatly in length, read- and write-set size, and degree of contention. RMS-TM applications also exercise a variety of special TM features, including nested transactions, I/O, and system calls and complex function calls inside transactions. Unlike STAMP, the RMS-TM suite was developed using the C++ TM standard rather than a library-level API. Running these

applications directly on the RSTM back ends, without the shim library, would have required large amounts of tedious and error-prone hand instrumentation. We report results for one application from each of the RMS-TM application domains: HMMcalibrate (from Bioinformation), UtilityMine (from Datamining) and Fluidanimate (from Physics).

We perform our experiments on a 2.27 GHz, 2-processor Intel Xeon (E5520) system. Each processor contains four hyperthreaded cores serviced by private 32KB L1 Icache and 32KB Dcache, a private 256KB L2 cache, and a shared 8MB L3 cache. The system is equipped with 8GB of RAM that each processor access through a QPI memory controller. Benchmarks are written using the subset of the C++ TM draft API [6] supported by the Intel® C++ STM Compiler Prototype Edition 4.0 [75], and compiled using −O3 settings. The reference input sets were used where applicable. Experiments were performed on Linux version 2.6.30. We rely on the default Linux thread scheduler which prefers to distribute threads across processors before cores before hyperthreads. The tested benchmarks and implementations do not benefit from hyperthreading, so we report results up to 8 threads only.

4.4 Experimental Results

4.4.1 Overhead Analysis of Automatic Instrumentation

While relying on a compiler to automatically instrument read and write accesses simplifies the instrumentation of complex programs relative to manual instrumentation, it may lead to over-instrumentation due to the need for conservative assumptions about aliasing and lack of idempotence. On the other hand, the compiler may identify optimization opportunities that were missed during manual instrumentation, therefore improving performance. To assess these potential effects, we compared the performance of the original, manually-instrumented STAMP applications to that of the automatically-instrumented versions that use our shim library. We could not perform the same analysis for RMS-TM, as manually instrumented versions are not available.

Linking with RSTM through the TM ABI shim library introduces some additional overhead, unrelated to the compiler, relative to manually instrumentation. We expect this overhead to be small—at most one additional function call per instrumented access.

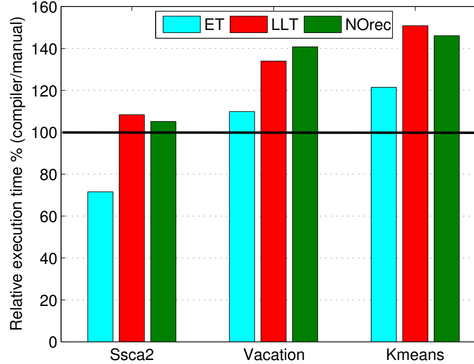


Figure 4.2: Execution time of compiler-instrumented code, relative to manually instrumented code, for single-threaded STAMP applications. Vacation represents "without waiver" execution.

As noted in Section 4.3, the manually instrumented versions of the `List` and `RBTree` data structures in Vacation use uninstrumented functions internally for frequently executed comparison operations. The compiler cannot possibly generate equally efficient code for these without a global understanding of the program, as the comparisons access shared memory locations. In the next section we provide Vacation results both with and without `__transaction [[waiver]]`. The former requires the same level of programmer understanding as the original implementation; the latter illustrates the overhead of leaving code generation entirely up to the compiler.

Figure 4.2 shows the overhead of automatic instrumentation for the single-threaded execution of the STAMP benchmarks (without `__transaction [[waiver]]`) on the three RSTM back ends. The results depend on both the applications and the back end. SSCA2 shows performance improvement for ET and limited overhead for LLT and NOrec. Since ET shows a net benefit, we believe that the compiler does a good job of instrumenting the code and identifying optimization opportunities, and that the different behavior of LLT and NOrec is specific to the STMs. For Kmeans and Vacation, on the other hand, all of the back ends suffer significant performance loss compared to the manually-instrumented version—from 10–50%. Here the compiler clearly introduces read/write instrumentation that the manually instrumented version was able to avoid, and extra optimization opportunities, if any, are insufficient to compensate.

Conservative instrumentation can have an effect on scalability as well. The resulting

Application	IntelSTM			NOrec			ET			LLT		
	2	4	8	2	4	8	2	4	8	2	4	8
HashTable	0.05	0.17	0.28	0.02	0.07	0.24	0.85	11.91	5.02	1.68	5.53	11.11
RBTree	0.00	0.00	0.01	0.00	0.00	0.00	0.02	0.01	0.09	0.18	0.48	1.57
DoubleList	13.85	36.31	52.13	10.09	27.56	49.48	7.75	29.15	57.35	14.81	37.38	63.16
SSCA2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.01	3.00
Kmeans	0.05	0.02	0.00	2.23	5.59	13.48	47.39	56.82	74.95	37.15	55.61	76.24
Vacation	0.01	0.04	0.08	0.00	0.00	0.00	0.80	1.13	4.26	0.08	0.26	0.66
HMMcalibrate	15.24	39.36	66.76	4.52	14.99	43.55	98.16	99.54	99.94	91.01	97.29	99.05
UtilityMine	0.01	0.05	0.26	0.00	0.03	0.10	0.09	1.44	0.80	0.11	0.41	0.93
Fluidanimate	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.04	0.02	0.05

Table 4.1: Abort Rates (percentage of all dynamic transaction instances that abort) for 2, 4 and 8 threads.

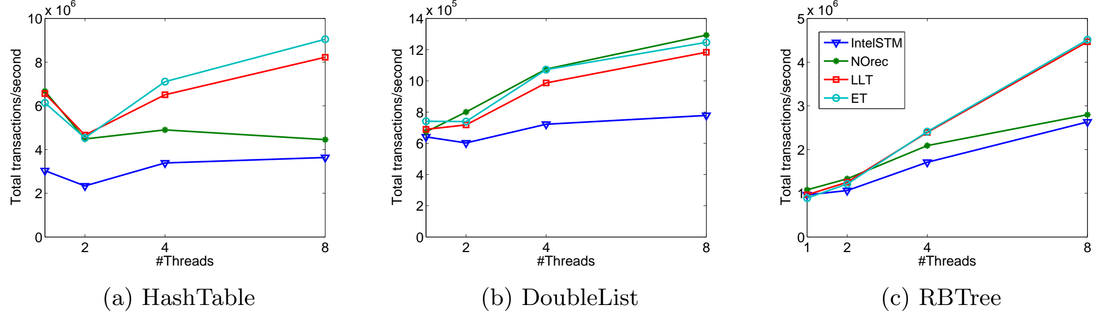
larger read and write sets lead to longer transactions, due to increased validation times and to an increase in the probability of false conflicts in orec-based implementations. In Kmeans, for example, manually instrumented code sees 8-thread abort rates for ET and LLT of 3% and 58%, respectively. For compiler instrumented code (Table 4.1), the corresponding rates are both around 75%. Clearly the extra instrumentation inserted by the compiler in this case interacts badly with eager conflict detection.

NOrec, which is lazy like LLT, sees an abort rate of approximately 14% with both manual and automatic instrumentation. It would be vulnerable, however, to increases in the number of instrumented writes, since its write-back operations are globally serialized. An even larger issue would arise in any application where compiler the instrumented writes in what could otherwise be a read-only transaction.

4.4.2 Back-end Comparisons

In this section we present performance results for the three sets of benchmarks mentioned in Section 4.3.

Microbenchmarks: In our first experiment we consider microbenchmarks in which a set of threads use transactions to continually insert, delete, and look up keys in a set. The set is prepopulated with half of the possible keys and we execute an instruction mix that consists of 33% of each operation. Approximately half of the insert and delete operations find the target key and modify the set, so transactions should be 66% read only. The IntelSTM compiler does not introduce any unnecessary writes, so the results presented here meet this goal. We consider three different set implementations—a hash table, a red-black tree, and a doubly linked list. Figure 4.3 reports the throughput



(a) HashTable (b) DoubleList (c) RBTree

Figure 4.3: Throughput results for the microbenchmarks. Y axis shows total number of transactions per second: higher is better.

(total number of transactions per second) for these microbenchmarks when varying the number of threads from one to eight.

In HashTable (Figure 4.3a) we test 8-bit keys (maximum set size of 256), and transactions are tiny, performing a maximum of five reads and three writes. This results in few conflicts, as seen in the low abort rates for all the back ends (between 0.24% and 11.11% with eight threads, as reported in Table 4.1). This configuration should be extremely scalable, however we immediately see the effect of Linux's default scheduling policy. Placing the second thread across the QPI interconnect results in long latencies and high overheads for data and metadata access once we have two threads. ET, LLT, and the IntelSTM can overcome this initial drop given enough threads, but NOrec's reliance on a single global sequence lock will not scale across the processors with such small transactions. Further investigation shows a high number of commit time re-validations for HashTable compared to the other microbenchmarks (23% of all commits with eight threads), which implies that NOrec transactions spend much of their time waiting in their commit barrier due to their need to validate after each writer commit. ET and LLT show better scalability at eight threads than IntelSTM; this may be attributed, at least in part, to the overhead of privatization safety in IntelSTM (not needed in the microbenchmark, and not provided by default in ET or LLT).

In DoubleList (Figure 4.3b) we again test 8-bit keys, but experience much more contention due to the linear structure of the list-based set. As with HashTable, DoubleList transactions perform a small number of writes, however they may perform up to 300 reads. These longer transactions reduce the relative overhead of metadata bottlenecks,

resulting in better scalability for the RSTM back ends. The large number of conflicts means that, in contrast to HashTable, ET and LLT validate nearly as frequently as NOrec. NOrec's higher throughput is a result of its lower abort rate, which stems in turn from value-based conflict detection and the resulting lack of false conflicts. It is currently unclear why larger transactions do not benefit IntelSTM as well. We suspect that contention management may play a role.

Finally, RBTree (Figure 4.3c) expands the key set size to 20 bits and illustrates the behavior of memory-bound applications. With set sizes approaching a million elements, RBTree transactions may perform over 100 instrumented reads and up to 50 writes during rebalancing. While data cache misses dominate execution time, ET and LLT scale better than IntelSTM and NOrec. In fact, IntelSTM and NOrec introduce larger runtime overhead because of the privatization-safe guarantees they provide. As with HashTable, NOrec's scalability is impacted by its need to validate when any writer commits.

STAMP: Figures 4.4a, , 4.4b, and 4.4c show performance results for the selected STAMP applications on the tested back ends.

Figure 4.4a shows Vacation results using the recommended "high" contention parameters, both with (dotted lines) and without (solid lines) `__transaction [[waiver]]`. With the waiver, Vacation exhibits large, read-dominated transactions—more than 1300 instrumented reads and 150 instrumented writes—with low contention, evidenced by low abort rates in Table 4.1. As expected, all back ends provide good scalability with performance improvement up to eight threads. Without the waiver, the number of instrumented reads roughly doubles, to more than 2500. IntelSTM continues to scale well in these conditions. The RSTM back ends, however, have a clear performance problem with read sets this large. As of this writing, the source of the problem is unclear, and is a subject of ongoing investigation. We would not have been aware of the issue without the availability of the Intel ABI to RSTM shim.

SSCA2 transactions (Figure 4.4b) consist of up to three reads and two writes, and are effectively independent of one another. Each transaction performs at least one write, and transactions form the bulk of application execution time. This represents the pathological workload for NOrec, where writer commits are serialized. We see this in NOrec's lack of scalability. In contrast, ET, LLT, and IntelSTM allow non-conflicting writers to commit in parallel and scale well. IntelSTM shows high overheads similar to

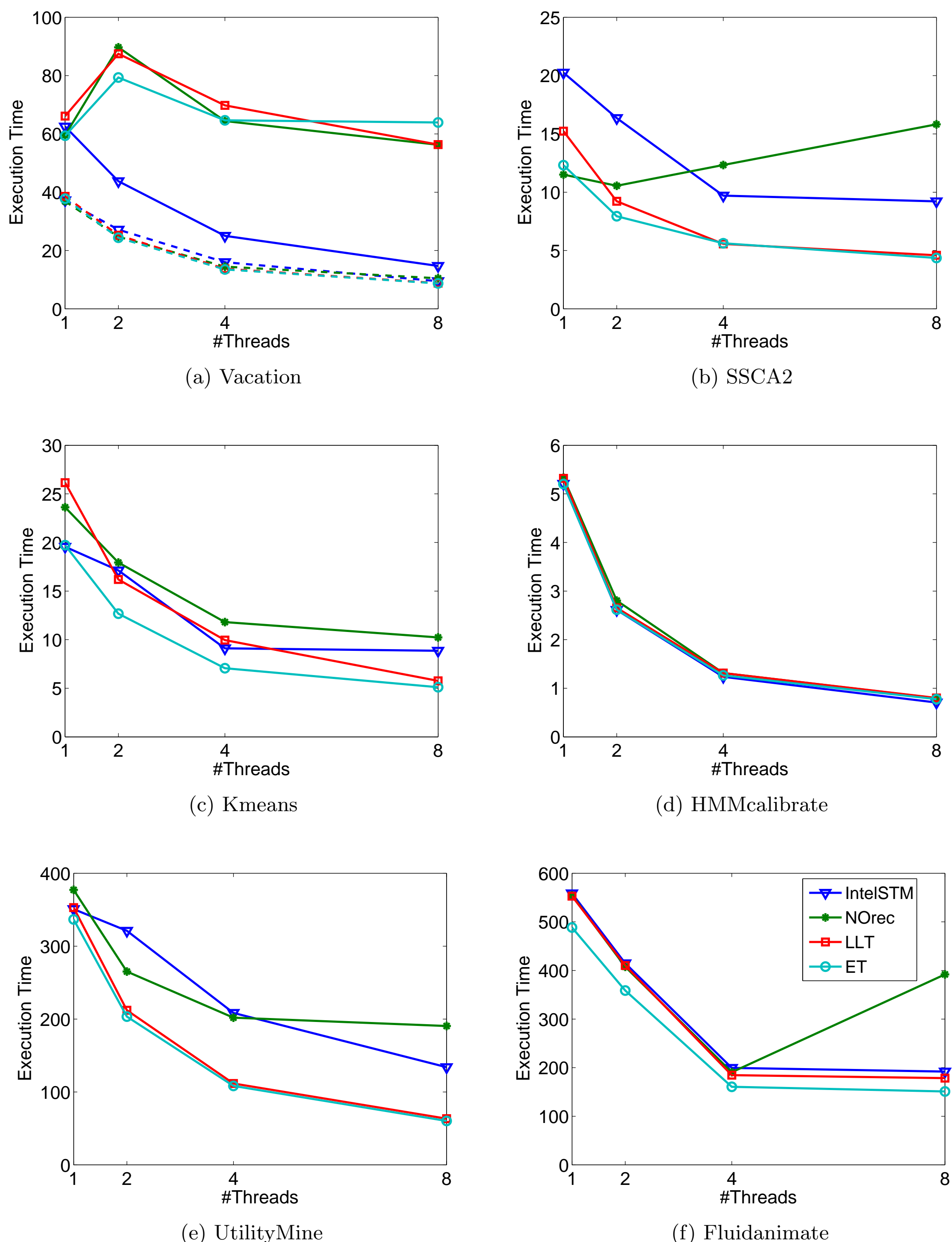


Figure 4.4: Scalability results for STAMP and RMS-TM. The Y axis shows execution time in seconds: lower is better. In Figure 5.5e, dotted lines represent the version with `__transaction [[waiver]]`

those seen in the HashTable microbenchmark, where transactions are similarly small and nonconflicting. We speculate that the cause of this overhead may be related to mechanisms used to provide privatization safety [121].

As discussed in Section 4.4.1, the Intel STM compiler appears to dramatically over-instrument Kmeans transactions. This results in larger read and write sets and, consequently, higher abort rates than those reported by Minh et al. [22]. For ET and LLT, the abort rates are particularly high: 75% or more at eight threads (Figure 4.1). The fact that NOrec sees only a 13% abort rate at eight threads suggests that most of the problem in ET and LLT is due to false conflicts. At the same time, compiler instrumentation results in all transactions being writers, which penalizes NOrec disproportionally, giving it the longest 8-thread execution time. Notice that, while the abort rate is very low with IntelSTM, its performance is similar to that of the other STMs. This suggests that its performance is dominated by other components.

RMS-TM: Figures 4.4d, 4.4e, and 4.4f show the execution time of the selected RMS-TM applications. HMMcalibrate exhibits short transactions with high contention. As shown in Table 4.1, it has the highest abort rate (between 44% and 99.9%, with eight threads). At the same time, it spends only a tiny fraction of its execution time inside transactions, allowing it to exhibit good scalability for all the back ends.

In UtilityMine (Figure 4.4e), IntelSTM shows high run-time overhead even with two threads. ET and LLT keep improving up to eight threads, but NOrec does not: a large number of threads increases the number of re-validations, leading to very little improvement beyond four threads. IntelSTM scales similarly to ET and LLT beyond two threads, but overall performance is dominated by the high instrumentation overhead.

Fluidanimate also has short transactions, but in contrast to HMMcalibrate, its contention is low. When increasing the number of concurrent threads, the number of transactions per thread remains constant, so the total number of transactions increases. On the other hand, the work done per thread decreases with the number of threads: as a result, Fluidanimate shows strong scalability up to four threads. With eight threads, however, the ratio between the computation and synchronization phases decreases, which limits scalability (Figure 4.4f). The high frequency of writer transactions (read/write ratio of 1.16:1) leads to a performance bottleneck in NOrec at eight threads.

Summarizing, our results show that scalability and overall performance depend heavily on both the application and the choice of back-end system. Generally speaking, high instrumentation overhead limits overall performance. IntelSTM shows significantly higher overhead than the RSTM back ends for some applications (e.g., HashTable and SSCA2). For these, even single-thread performance is significantly lower than with the other STMs. If the abort rate is high, value-based conflict detection (NOrec) helps reduce false conflicts and, therefore, improves performance. DoubleList and HMMcalibrate illustrate this effect. When the read/write ratio is low (i.e., the application has multiple active writer transactions), STMs that allow concurrent writers (ET, LLT and IntelSTM) show higher performance compared to single-writer STMs. We can see this effect strongly in SSCA2, and to a lesser extent in HashTable and Kmeans.

4.5 Related Work

Intel STM [161] consists of a C/C++ compiler and a high performance STM Runtime Library. The compiler instruments all shared memory reads and writes inside transactions by using read and write barriers. The flattening model is used to support nested transactions, and weak isolation between transactional and non-transactional code is provided. Transactions can be executed in optimistic or pessimistic mode. In both cases, the transactional writes update the data in-place with strict two-phase locking, while the transactional reads are executed optimistically or pessimistically. Serial execution mode is also provided to support transactions that contain irrevocable operations.

GCC-TM [138] is focused primarily on the challenges of programming in the large. One major issue is the need to maintain two or more copies of certain functions: the traditional version for use outside transactions, and one or more versions with instrumented reads and writes, for use inside transactions. Among other things, linkers and object file conventions must be extended to accommodate the multiple versions, and to identify which to call on any given code path. In a similar vein, support must be provided for programmer annotations (attributes) [6], which can dramatically increase the number of functions that are callable within transactions—or their expected performance when called. Comparatively little attention is currently being paid to the STM

back end for GCC, which is a simple, correct, but unoptimized implementation based loosely on the TinySTM of Riegel et al [133].

Bartok [68] is an ahead-of-time C# compiler which has language-level support for TM. 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.

Tanger [53] is an extension for the LLVM [94] compiler framework that automatically transactifies applications. The programmer only has to mark the start and the end of the transactions. The instrumentation delegates all shared data accesses in these regions under the control of the TinySTM [133] library.

In comparison, the RSTM package includes several STM run-times, each of which appears well suited to some workloads. The most widely held conclusion from STM research so far is that no one runtime is ideal for every application. That's why we believe that the various RSTM back ends provide richer variation in terms of STM algorithms, and much more opportunity to tune the system to application needs.

4.6 Conclusions

As Transactional Memory moves towards a more robust and mature stage, it becomes essential to be able to share and run applications, compilers, and run-time systems among groups. Standardization is a key step in this direction. However, while releases of compilers with support for TM are available, much of the work that has been done on STM runtimes is not compatible with those compilers, because of interface issues.

In this chapter we described work that makes back ends from the RSTM suite (specifically, LLT, ET and NOrec) compatible with the Intel TM ABI, and with compilers that conform to that ABI. This work entailed modest changes to RSTM itself, plus the creation of a shim library that adapts the Intel ABI to the RSTM API. Using the newly available back ends, we evaluated the performance of several applications from the STAMP and RMS-TM benchmark suites; the former required manual re-writing to eliminate the manual instrumentation of the STAMP API and to accommodate the need for annotations on functions called within transactions in the C++ TM API.

Our work makes it possible, for the first time, to run large applications from other groups on the RSTM back ends, and to obtain an "apples to apples" comparison of

back ends using such applications. It also allows us, in the case of STAMP, to compare automatically and manually instrumented applications.

We find that memory footprint, abort rate, and consequent performance depend heavily on both the particular application and the choice of back-end system. This result confirms earlier findings with microbenchmarks; from it we conclude that diversity in back ends is essential, and that dynamic adaptation among back ends (as explored, for example, by Spear [151]) is a promising research direction. Our experiments also show that, while unnecessary instrumentation introduced by the compiler may induce considerable run time overhead, the Intel STM compiler is able to exploit optimization opportunities that may actually improve performance over hand-instrumented code in certain cases.

Part III

Design and Implementation of a High Performance STM

Performance and scalability of current TM designs on aggressive multi-core/multi-threaded processors do not always meet the programmer's expectation. This is especially true for STM designs, where the overhead of instrumentation and transactions' management severely limits application's performance, especially at large scale.

We propose a new STM design (Chapter 5), STM^2, based on an assisted execution model in which time-consuming TM operations are offloaded to *auxiliary threads* while *application threads* optimistically perform computation. Surprisingly, our results show that is often more convenient to use additional processing elements to support computation rather than performance computation: STM^2 provides, on average, speedups between 1.8x and 5.2x (and up to 12.8x) over state-of-the-art STM systems.

On the other hand, assisted-execution systems may show low processor utilization. In order to alleviate this problem and to increase the efficiency of STM^2, we used an integrated hardware/software approach to dynamically partition hardware resources at fine-grained level (Chapter 6). We enriched STM^2 with a runtime mechanism that automatically and adaptively detects application and auxiliary threads' computing demands and dynamically partition hardware resources between the pair. This dynamic mechanism further improves STM^2's performance (up to 85% over the standard STM^2 design) and efficiency.

Chapter 5

STM^2: A Parallel STM for High Performance SMT Systems

5.1 Introduction

Performance and scalability of current TM systems is not always satisfactory, especially for STM proposals where the overheads introduced by the STM runtime system may well outweigh the parallelism gained [24]. Performance of several applications among the most common benchmarks suites, such as STAMP [22], are limited by STM overhead and provide performance degradation beyond a certain number of threads [24]. Some authors report drastic slow-downs when using STM (e.g., only breaking even with optimized sequential code after using 8 cores [24]). Even state-of-the-art TM systems typically require at least two threads to achieve performance that matches optimized sequential code [39, 68].

In this chapter we propose a novel, high-performance STM design that provides higher performance and scalability for real TM applications. Our design is based on two main observations: First, the TM system itself may limit scalability by introducing run time overhead or serializing the the execution of the application's threads to resolve conflicts. In particular, STMs that use bookkeeping introduce considerable slowdown due to read- and write-set validation and transaction state management [24]. Moreover, the STM needs to correctly handle conflicts among transactions running concurrently, eventually aborting and rolling back one or more transactions. Second, in any parallel program, Amdahl's law [9] limits the extent to which parallel execution can achieve

speedups. With TM, if large sections of parallel code run within transactions, there is a risk that the speedup possible via Amdahl's law will never be enough to recover the overheads of using TM. On the other hand, 4- or 8-core architectures with tens of hardware threads are already available [112, 162] and this count is expected to increase in the coming years, thus scalability of parallel applications is crucial for extracting high performance from future multi-core and multi-threaded systems.

Both STM runtime overhead and the Amdahl's Law limit on the theoretical speedup suggest the use of assisted execution models [47], in which some of the computing elements (cores or hardware threads) are used to support computation rather than being devoted to running additional application threads [98, 109, 166]. The intuition behind assisted execution models is that some of the computing elements can accelerate sequential part of the application and/or relieve application threads from handling runtime functionalities, therefore pushing further the theoretical Amdahl's Law's speedup and reducing runtime overhead. Since not all applications are able to effectively use all cores and/or hardware threads, we propose to perform time-consuming STM operations on those computing elements that do not provide measurable performance improvement. The general idea is that using additional cores/hardware threads to speed up STM operations may provide higher performance than using these processing elements to run additional application threads. Specifically, we offload read-set validation, bookkeeping, transaction state management and conflict detection to an *auxiliary thread* running on a *sibling* core/hardware thread, i.e., a processing element that shares some levels of hardware resource (like the L1 or L2 cache) with the *application thread*.[1]

In order to demonstrate our proposal we implemented *Software Transactional Memory for Simultaneous Multithreading* systems (STM^2 - pronounced *STM-squared*). To the best of our knowledge, STM^2 is the first parallel STM system that uses secondary hardware threads to leverage STM overhead. STM^2 is essentially a parallel STM system where transactional operations are divided between *application threads* (computation) and *auxiliary threads* (STM management). With STM^2, application threads optimistically perform their computation with minimal support from the underlying STM system. All synchronization and STM management operations are performed by the

[1]Two hardware threads in a core or two cores sharing the L2 cache are examples of sibling hardware thread/core, respectively.

paired auxiliary threads. This means that application threads experience minimal overhead. Auxiliary threads, instead, validate read-sets, maintain transaction states and detect conflicts in parallel with the application threads' computation. STM^2 detects conflicts as soon as they occur (eager conflict detection). If a conflict is detected, the auxiliary thread interrupts its corresponding application thread and aborts the transaction. If no conflicts arise during a specific transaction, the auxiliary thread commits the transaction and updates the modified shared memory location (lazy update). Communication between application threads and their corresponding auxiliary threads is performed through a lock-free circular buffer and simple atomic state variables.

We tested STM^2 on an aggressive, high performance SMT processor, an IBM POWER7 system with a total of 32 hardware threads. To the best of our knowledge, this is the first study that tests transactional memory on a POWER7 processor. Our results show that by overlapping computation and STM management operations STM^2 obtains performance improvement, outperforming modern well-known STM systems, namely TinySTM, NOrec, TML and TL2, for several STAMP benchmarks. Our experiments show that STM^2 achieves, on average, between 1.8x and 5.2x speedups over state-of-the-art STM systems, with peaks up to 12.8x.

This work makes the following contributions:

- Introduces STM^2, a novel parallel STM implementation that reduces the runtime overheads by offloading time consuming TM management operations to auxiliary threads running on sibling hardware threads.

- Tests several state-of-the-art STM systems, namely TinySTM, TL2, NOrec and TML, on an aggressive multithreading processor designed for high performance.

- We show that, perhaps surprisingly, it is often better to use hardware threads to parallelize the STM implementation, than to devote those hardware threads to running additional application threads.

The rest of this chapter is organized as follows: Section 5.2 motivates our work. Section 5.3 describes the design of STM^2 and provides internal details of the implementation. Section 5.4 and Section 5.5 describe our experimental setup and results, respectively. Section 5.6 summarizes related work. Finally, Section 5.7 concludes this work.

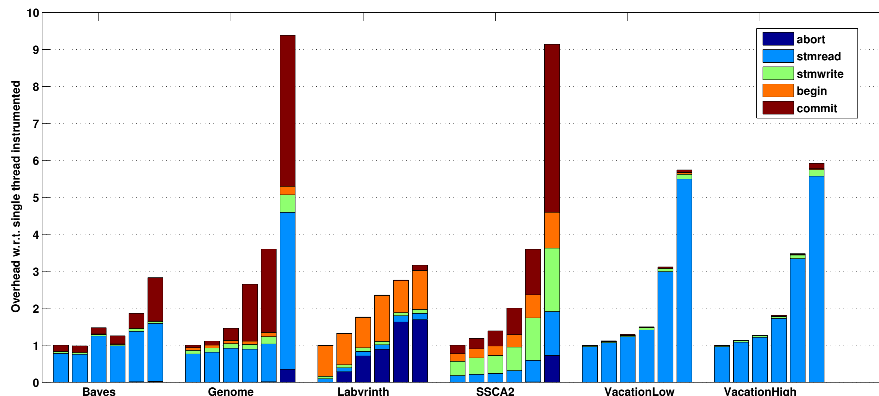


Figure 5.1: TinySTM per-transaction overhead breakdown for STAMP applications with respect to instrumented single thread version. We report per-transaction overhead because the number of transactions per thread for STAMP applications decreases with the number of threads. We instrumented TinySTM and obtained per-transaction overhead breakdown for single thread, 2, 4, 8, 16 and 32 concurrent threads versions. STM per-transaction overhead increases with the number of threads because of the higher pressure on internal STM data structure, more frequent read-set validations and higher contention.

5.2 Motivation

STM management operations are time-consuming and may introduce considerable overheads that increase with the number of threads running in parallel and the size of the read- and write-set [86]. The result is that STM systems may not be able to provide satisfactory performance at scale [24]. In order to understand the overhead introduced by STMs, we run preliminary experiments instrumenting TinySTM, a widely used STM system. Figure 5.1 shows the per-transaction overhead introduced by TinySTM on STAMP benchmarks running on an IBM POWER7 system when varying the number of threads from 1 to 32.[1] Since the total number of transactions in STAMP benchmarks is constant, the number of transactions per thread decreases with the number of concurrent threads. We, hence, report the per-transaction overhead of each benchmark normalized to the overhead introduced by the STM when running the same application with one thread. For example, *VacationLow* presents large, mostly-read transactions,

[1] In these experiments and in the rest of the chapter, we use the default configuration for TinySTM, eager conflict detection and lazy data versioning.

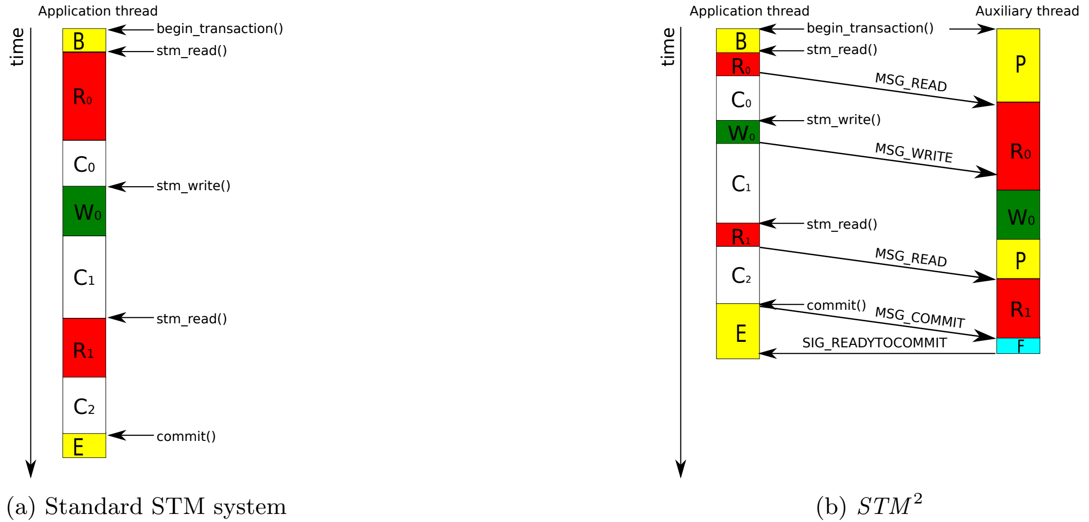
therefore, each transaction spends most of its time in the `stm_read()` function. When going from 1 to 32 concurrent threads, the commit rate increases and more read-set validations need to be performed. For the specific case of *VacationLow*, the STM overhead with 32 concurrent threads increases by 5.7x with respect to the instrumented single thread execution. While *VacationLow*'s and *VacationHigh*'s transactions are dominated by read time, *Genome* and *SSCA2* have short transactions, thus their per-transaction overhead breakdown is completely different. In particular, Figure 5.1 shows that the relative overhead of `begin_transaction()` and `commit()` have a higher impact on *SSCA2* than on *VacationLow* in the instrumented, single thread version. Figure 5.1 also shows that for applications with high contention, such as *Labyrinth*, the overhead introduced by aborts increases at scale.

Our experiments, in accordance to what was previously observed [24, 86, 108], show that per-transaction STM overhead increases with the number of concurrent threads, which may limit scalability substantially. Note that the actual impact on the applications' performance may vary depending on the amount of time each application spends inside transactions. Section 5.5 discusses the impact of STM overhead on each application's performance.

5.3 STM^2 Design and Implementation

Recent studies [89] show that future scientific problems with large data sets will require higher computational power (i.e., higher number of cores) than what is currently available. Processing elements that do not directly provide performance improvement should be used in a better way, for example, to leverage the work performed by overloaded cores. STM^2 is a novel implementation that goes in this direction: time-consuming operations (such as read-set validation and conflict detection) are offloaded to *auxiliary threads* running on separate hardware threads. Application threads have fewer STM management operations to perform and can spend their cycles on more useful work.

Figure 5.2b shows how offloading operations to auxiliary threads may reduce STM overhead, therefore improving performance. Figure 5.2a shows how a typical eager conflict detection, lazy data versioning STM system works. Before actually accessing any memory location, the application thread performs an `stm_read()` when reading a memory location, or an `stm_write()` when attempting to modify a shared variable.



(a) Standard STM system

(b) STM^2

Figure 5.2: STM^2 offloads time-consuming STM operations to sibling hardware threads. In this Figure the application thread performs two read (R_0 and R_1) and a write (W_0) operations. C denotes computational phases that do not access shared memory locations. `begin_transaction()` and `commit()` are marked with B and E, respectively, while P denotes polling and F denotes commit phase at the auxiliary thread's side. Offloading STM operations to auxiliary threads reduces the overall execution time.

These two functions notify the STM runtime about which locations should be inserted into the read-set and the write-set of that thread. Whenever the STM runtime system takes control, it may check whether a conflict has occurred and, if so, abort a conflicting transaction. As Figure 5.2a shows, the application thread often runs STM library code rather than performing its computation, especially if the STM operation triggers time-consuming activities, such as read-set validation.

We propose to move time-consuming STM operations to another hardware thread and perform them in parallel with its application thread. Figure 5.2b shows our approach: Whenever an application thread accesses a memory location (either reading or writing), it simply sends a message to its corresponding auxiliary thread and then keeps performing its computation. The auxiliary thread, in turn, waits for messages coming from its corresponding application thread and performs read-set validation, transaction state management and conflict detection. Whenever an auxiliary thread detects a conflict, it aborts its corresponding application thread. As Figure 5.2b shows, offload-

ing STM operations to auxiliary threads and performing transaction management in parallel reduce the transaction's execution time, therefore improving performance.

STM^2 is an eager conflict detection, lazy update STM system. Given that we are using an auxiliary thread that runs in parallel with the application thread, it makes sense to perform as many operations as possible in parallel. In this scenario, lazy conflict detection would delay most of the work at commit stage, serializing the execution of the application thread (that would mainly run during the transaction) and the auxiliary thread (that would be idle during the transaction and overloaded at commit time) and indeed invalidate most of the benefit of our approach. On the other hand, the main drawback of eager conflict detection is the extra overhead caused by the STM management operations (Figure 5.2a). This overhead is exactly what STM^2 reduces. Eager conflict detection increases the parallelism of STM^2 and decreases the amount of work to be done at commit stage, which is a synchronization point and critical for the STM^2 performance (see Figure 5.2b). Eager updates, instead, would require extra communication among the auxiliary threads. Memory locations modified by aborted transactions must rollback to their original values, hence, all transactions that have read those invalid values may have to rollback too. In STM^2, memory updates and aborts are handled by auxiliary threads, hence they should also take care of restoring memory locations modified by aborted transactions. This, in turn, would require auxiliary threads to exchange messages among themselves. Although eager updates is a possible solution, lazy updates minimize communication among auxiliary threads.

Offloading time-consuming STM operations to a secondary processing element is particularly appealing for multithreading architectures (like IBM POWER, Intel with Hyper-Threading or SUN Niagara). In STM^2, application and auxiliary threads are paired on the same core, i.e., they are pinned to two separate hardware threads of the same core. While application and auxiliary threads could run on different cores, running on the same core is advisable for the following reasons: 1) the cost of a hardware thread (in terms of space, resources and power consumption) is lower than that of a core; 2) even though extra cores may improve performance linearly, extra hardware threads usually provide only between 1.2x and 1.6x speedup [4], and 3) application and auxiliary threads running on the same core share more resources (for example, the L1 cache), which allows lower-latency communication.

In the current implementation, STM^2 supports a basic TM programming model in which a transaction that aborts does not necessarily see a consistent view of memory, and in which there is no conflict detection between transactional and non-transactional memory accesses. Consequently, the programmer or compiler using STM^2 must sandbox the effects of "zombie" transactions, and must ensure that data is accessed in a consistent way (e.g., using the fence techniques of Spear et al. [148], or using the memory protection isolation mechanisms of Abadi et al. [2]). This is the programming model typically used in STAMP and other TM applications (e.g., *Labyrinth* explicitly restarts inconsistent transactions) therefore no extra support is required to run STAMP benchmarks.

The following subsections describe with more detail the main components of STM^2: application and auxiliary threads synchronization (Section 5.3.1), transactional write (Section 5.3.2) and read (Section 5.3.3) operations.

5.3.1 Application/Auxiliary Thread Synchronization

Application and auxiliary threads communicate through a communication channel and atomic status variables. Application threads send messages to their paired auxiliary threads to notify read and write operations. These operations require extra parameters and cannot be implemented by a simple shared variable (see in following subsection). Auxiliary threads, instead, only need to send two signals[1] to application threads: `SIG_READYTOCOMMIT` and `SIG_ABORT`. We thus implemented a single-producer/single-consumer, circular, lock-free queue where the application thread (producer) posts read and write messages that the auxiliary thread (consumer) retrieves and processes. The `SIG_READYTOCOMMIT` and `SIG_ABORT` signals do not need extra information and are implemented through atomic status variables shared between application and auxiliary threads. These variables are accessed and modified using atomic operations. An extra signal (`SIG_START`) and message (`MSG_COMMIT`), are sent to auxiliary threads when a transaction begins or ends. When an application thread is not involved in a transaction, its corresponding auxiliary thread waits in a spinning loop. As the application thread enters a transaction (`begin_transaction()`), the auxiliary thread receives the `SIG_START` signal and starts polling the communication channel for incoming messages. When an application thread reaches the end of a transaction and attempts to commit

[1]Note that these are different from operating system signals.

(`commit()`), it sends the `MSG_COMMIT` message and waits for the auxiliary thread to complete its work by spinning on the `SIG_READYTOCOMMIT` signal. If the auxiliary thread succeeds resolving all conflicts and validating its read-set, it commits the transaction by updating all shared memory locations modified by the application thread and sets the `SIG_READYTOCOMMIT` signal.

Finally, all shared atomic variables are modified only by one of the two threads during a transaction. For example, auxiliary threads set `SIG_READYTOCOMMIT` and `SIG_ABORT` signals while application threads only read the status of these variables. Similarly, only application threads set the `SIG_START` signal, while auxiliary threads only poll on its value. The result is that the communication involved is minimal and we believe that a small extra hardware buffer between two hardware threads may eliminate the need of using the L1 and increase performance.

5.3.2 Writing to a shared memory location

Memory locations modified by application threads during transactions are not visible to other threads until the transaction commits. On the contrary, conflicts are detected as soon as they occur, avoiding unnecessary computation for transactions that will be aborted and reducing the overhead at commit time.

To guarantee correctness, only one application thread at a time is allowed to change the value of a particular shared memory location, although several threads can modify different memory locations at the same time. Before altering a memory location, application threads need to be sure that no other thread is currently attempting to modify the same location. STM^2 uses *ownership records* to identify which thread is entitled to change the value of a given shared memory location. Once a thread owns a location, it is allowed to modify its content. Any other thread that needs to alter the content of the same location and, therefore, tries to acquire its ownership, will fail (conflict) and will restart the transaction. STM^2 maintains a per-thread *write-set* buffer to temporarily store values modified during a transaction but not yet committed. If the transaction commits successfully, STM^2 will publish its write-set. The updated values will then become visible to the other application threads. STM^2 uses versioning based on extendable timestamps to detect conflicts [133]: every time a shared memory location is updated with a new value, the current timestamp is used as version number and associated with that location. A conflict arises when an application thread has

```
void stm_write(Addr, Val)
{
  if (writeset.insert(Addr, Val)
       ==
     present)
     return;
  else
     channel.send(
         MSG_WRITE, Addr, Val);
}
```

(a) Application thread transactional write

```
void aux_stm_write(Addr, Val)
{
  if (validate() == fail) abort();
  if (acquire_ownership(Addr))
      ownedlist.add(Addr);
  else
      ret = cm.onconflict();
      if (ret == CM_ABORT)
         abort();
}
```

(b) Auxiliary thread transactional write

Figure 5.3: Pseudo-code for application and auxiliary thread STM write

read a value from a memory location whose version number is lower than the current one.

Whenever an application thread wants to modify a shared memory location, it issues an `stm_write()` call, passing the address of the memory location and the new value as arguments. Figure 5.3 shows the pseudo-code for `stm_write()` on both application and auxiliary thread sides. On the application thread side (Figure 5.3a), `stm_write()` checks whether the location is already in the write-set, in which case the application thread simply updates the value and returns. If the location is not in the write-set, the application thread will still optimistically write the new value to its write-set but it will also send an `MSG_WRITE` message to its corresponding auxiliary thread. Upon receiving an `MSG_WRITE` message, the auxiliary thread first validates its read-set and then tries to acquire the ownership of the target memory location (Figure 5.3b). If both operations are successful, the auxiliary thread adds the location to its list of owned shared memory locations. At commit stage, these locations will be updated in memory and the new values will become visible to the other application threads. Note that, on success, no other message is sent to the application thread because it had optimistically already proceeded with the transaction. If the auxiliary thread detects a conflict while trying to acquire the ownership of the location, the contention manager will decide which transaction has to abort. In case the contention manager returns `CM_ABORT`, the auxiliary thread notifies its corresponding application thread by setting the status of

```
void stm_read(Addr)
{
  found = writeset.find(Addr);
  if (found)
    return from writeset;
  else
    channel.send(MSG_READ, Addr);
    return from memory;
}
```

(a) Application thread transactional read

```
void aux_stm_read(Addr)
{
  if (is_owned(Addr))
     ret = cm.onconflict();
     if (ret == CM_ABORT) abort();
  if (validate() == fail)
     abort();
  else
     reads.insert(Addr);
}
```

(b) Auxiliary thread transactional read

Figure 5.4: Pseudo-code for application and auxiliary thread STM read

the transaction to aborted. The auxiliary thread then removes all entries in the read-set, releases all owned locations, and rolls back. Whenever an STM operation is issued, application threads check their transaction's status and restart the transaction if they find out that the transaction has been aborted by their paired auxiliary threads. Note that, besides resetting the write-set, no other actions are required from the application thread on abort.

We minimized synchronization overhead by using a lock-free data structure for the communication channel described in Section 5.3.1, and by clearly dividing data structures between application and auxiliary threads. Auxiliary threads own the read-set and the list of owned locations. Application threads, on the other hand, own the write-set. Since application threads never access auxiliary threads' data structures (and vice versa) there is no need to protect them with locks.

5.3.3 Reading from a shared memory location

Application threads read shared memory locations by calling the `stm_read()` function and passing the address of the target memory location as argument. The `stm_read()` has three main goals: 1) locate the current version of the shared value to return, 2) insert the address of the shared location in the transaction's read-set (unless it is already present), and 3) perform read-set validation, if required. These operations are divided between the application and the auxiliary threads.

Figure 5.4 shows how application and auxiliary threads operate when reading a memory location. The application thread (Figure 5.4a) locates the current version of the value to be read. The current value is either stored in the transaction's write-set or in the original memory location. In the former case, the application thread has already issued at least one write operation on that location at the time of reading the value. In this case STM^2 returns the value modified by the last write operation contained in the transaction's write-set. If the address is not found in the write-set, STM^2 returns the current version from memory and sends an `MSG_READ` message to the auxiliary thread together with the address of the target memory location. Note that other threads may be modifying the same memory location but those threads have not committed their transactions yet, hence those modifications are not visible to the current thread.

When the auxiliary thread receives the `MSG_READ` message, it performs conflict detection. A conflict occurs when 1) the memory location is locked by another thread or 2) the version read by the application thread is different from the current version, i.e., some other thread has committed a new version (`validate()` returns `fail`). In the former case, the auxiliary thread calls the contention manager which may decide to abort either the current transaction or the one that has locked the location. In the latter case, the transaction aborts. If no conflicts are detected, the auxiliary thread inserts the memory location's address into the read-set and moves to the next message.

5.4 Experimental Setup

This section describes the setup environment, the benchmarks and the STM systems used in our experiments.

We performed our experiments on an IBM POWER7 [4, 162], an out-of-order, 8-core design where each core is 4-way SMT (32 hardware threads in total). Each core region (or "chiplet") contains a 32 KB 4-way set associative L1 I-cache and a 32 KB 8-way set associative L1 D-cache, a private per-core 256 KB L2 cache and a 4 MB portion of the shared 32 MB L3 cache. Since POWER7 is capable of running 32 threads concurrently, we limit our experiments to 32 threads without over-provisioning the system (i.e., we run as many threads as available hardware threads). Each POWER7 core can run in single-thread (ST) mode, SMT2 (two threads executing on a core concurrently) or SMT4 (three or four threads executing on a core) mode. For capacity computing (i.e.,

multiple independent, serial jobs running in parallel), both SMT2 and SMT4 modes are expected to provide benefits. For capability computing (i.e., parallel applications with high degree of parallelism), SMT4 may not show extra benefits [4]. STM^2 uses the SMT4 mode and offloads time-consuming TM operations to secondary hardware threads that, otherwise, may not provide extra performance improvement.

We compare STM^2 to several well-known, publicly available and mature STM proposals, namely TML [149], NOrec [39], TinySTM [133], and TL2 [43], using the STAMP benchmark suite [22] compiled with gcc 4.3.4 and `-O3` settings. **TML** is an eager conflict detection, eager versioning system with a single sequence lock [92]. **TL2** is a lazy conflict detection, lazy versioning system. **TinySTM** is an eager conflict detection, lazy versioning system with extendable timestamps. **NOrec** extends TML with lazy updates and value-based conflict detection.

We selected these STM systems because they reflect popular but divergent points in the STM design space. Several of these STM systems have not been officially ported on POWER architectures (e.g., TL2, NOrec). We ported those STM systems on POWER processors[1] to be able to fairly evaluate STM^2 but some of the STAMP benchmarks (namely *Intruder*, *Kmeans* and *Yada*) did not execute correctly with some of the tested STMs due to bugs in STAMP code [25]. We omit these results for those benchmarks for fairness. Finally, in order to evaluate the effect of increasing the read-set size on the performance of the STMs, we run two versions of *Vacation* (i.e., *VacationLow* and *VacationHigh*).

5.5 Experimental Results

In this Section we analyze the performance of STM^2 and the other tested STM systems. Figure 5.5 shows performance of STAMP benchmarks running on the IBM POWER7 system previously described. We report the execution time of each STAMP benchmark when varying the number of threads from 2 to 32. In the first set of experiments, we compare STM systems running STAMP benchmarks when using the same number of application threads: we, thus, compare STMs with N threads to STM^2 running N application threads plus N auxiliary threads (N+N), for N=2, 4, 8, 16. While in these experiments STM^2 uses double the number of threads (N+N) than the other STMs

[1] No further modifications to the original implementations have been applied.

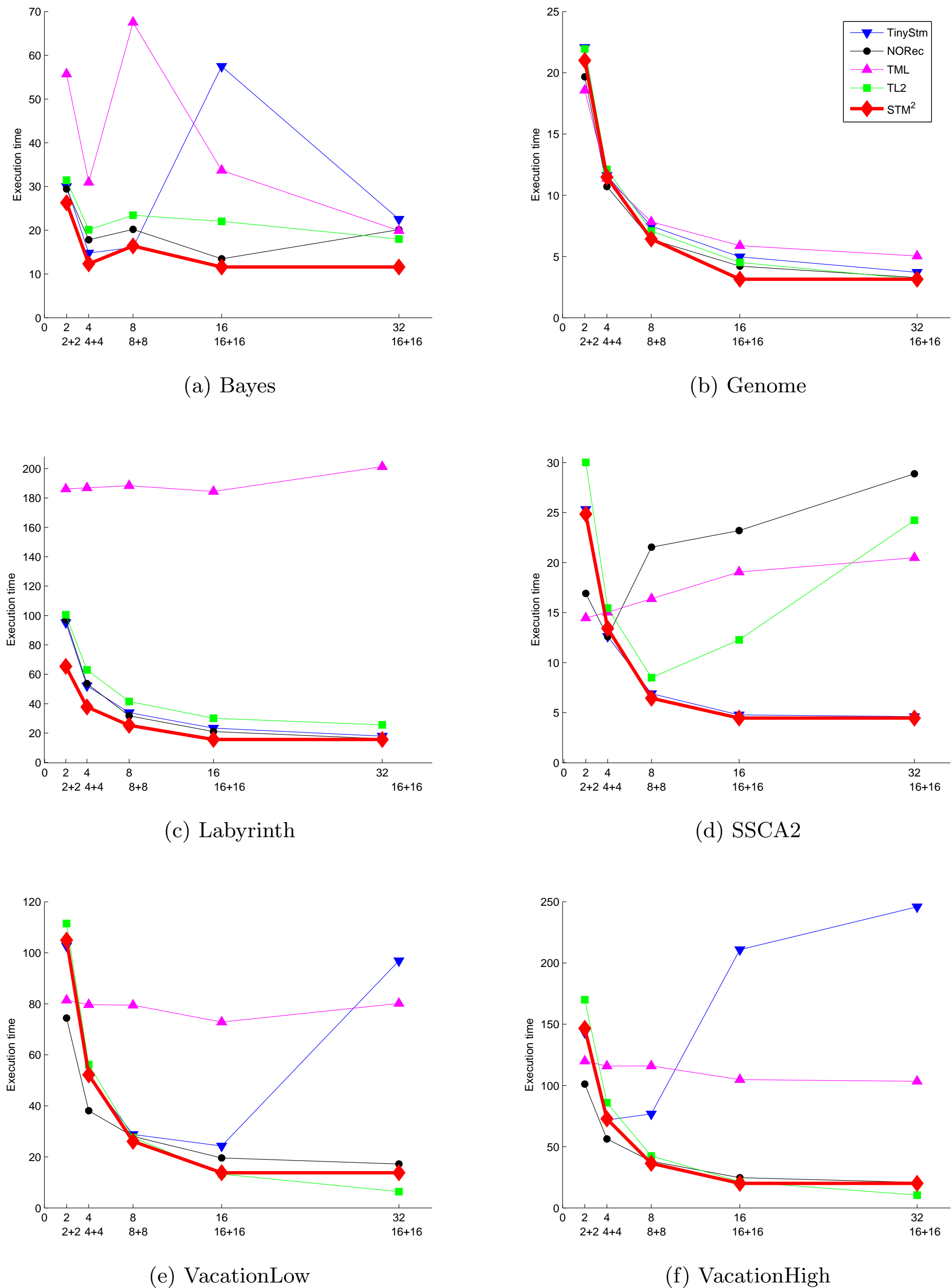


Figure 5.5: STAMP benchmarks with different STMs. The x-axis reports the number of used threads, which is N for the standard STMs and N+N for STM^2, for N=2,4,8,16. For N=32, we compare STMs performance to STM^2 using 16+16 threads (we repeat this value in correspondence of N=16 and N=32 to facilitate comparison with the other STMs having equal hardware resources). In the graphs, lower is better.

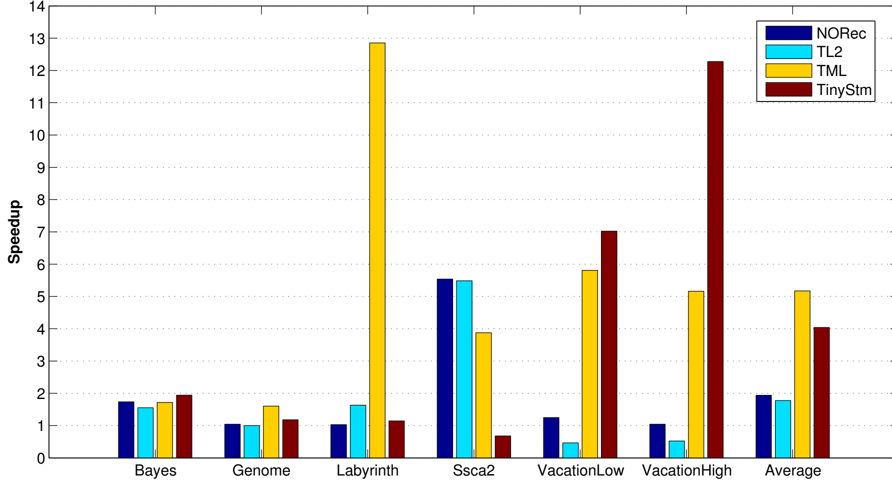


Figure 5.6: Speedups of STM^2over tested STMs for STAMP applications using the same amount of hardware resources (32 hardware threads).

(N), the extra hardware threads are available and there is no reason why they should be left idle if an STM can take advantage of them. Moreover, in this set of experiments, the number of transactions per application thread is the same. In the second set of experiments, we analyze the performance of STM^2 and the other STMs using the same amount of hardware resources: we compare STM systems with 32 threads to STM^2 with 16 application threads and 16 auxiliary threads (16+16). We report STM^2 speedups for this experiment (32 threads versus 16+16 threads) in Figure 5.6.

As we can see from Figures 5.5 and 5.6, STM^2 reduces runtime overhead by offloading time-consuming operations to dedicated hardware threads. The reduced overhead directly translates to better performance (lower execution time).

TinySTM Both TinySTM and STM^2 use eager conflict detection and lazy versioning. TinySTM is, thus, the ideal STM system to be compared with in order to analyze the effect of offloading transaction state maintenance, read-set validation and conflict detection to secondary hardware threads. As Figure 5.5 shows, STM^2 performs better or equal than TinySTM in all cases. If the level of contention is low and the read-set size are small (*Genome* and *SSCA2*), STM^2 and TinySTM behave similarly, especially at small scale (N=2 or N=4 threads). When the read-set becomes larger, STM^2 clearly outperforms TinySTM. For example, STM^2 performs consider-

ably better when running *VacationLow* (7x faster) and *VacationHigh* (12.3x faster) with 32 hardware threads. *VacationLow* exhibits large, mostly-read transactions, thus its read-set size is considerably larger than other applications. Eager conflict detection requires scanning read-sets to identify possible conflicts during the execution of each transaction. In this scenario, larger read-sets introduce higher runtime overhead. Moreover, as reported by Cascaval et al. [24] and confirmed by our experiments (Figure 5.1), runtime overhead increases with the number of concurrent threads. STM^2 is able to absorb transaction state maintenance, read-set validation and conflict detection overheads with the secondary hardware threads. In our experiments, TinySTM is not always able to scale beyond N=16 threads: *VacationLow* takes about 24.24 seconds with N=16 threads and 96.88 seconds with N=32 threads. STM^2 instead is able to make a better use of the last 16 hardware threads by accelerating STM operations and reducing the execution time to 13.79 seconds (7x faster) when using 16 applications threads and 16 auxiliary threads (16+16). Moreover, STM^2 is also faster than the best TinySTM performance obtained with N=16 threads (1.8x). We conclude that the STM overhead introduced by TinySTM on *VacationLow* is completely absorbed by the auxiliary threads in STM^2. The effects of offloading STM operations to secondary hardware threads become more evident when increasing the number of read operations performed during each transaction or the level of contention in the application. *VacationHigh* performs the same algorithm as *VacationLow* but its transactions operate on more items (i.e., larger read-sets). Figure 5.5f shows that TinySTM does not provide performance improvement beyond N=4 threads (in fact, performance constantly reduces with the number of threads). STM^2, instead, efficiently scales up to 32 hardware threads (16+16), providing a final speedup of 12.3x over TinySTM with 32 hardware threads.

Bayes and *Labyrinth* exhibit a high level of conflict, even though their read- and write-sets are not as large as in *VacationLow*. STM^2 performs better than any other STM in these two cases and, in particular, shows a 1.9x and 1.1x speedup over TinySTM with 32 hardware threads for *Bayes* and *Labyrinth*, respectively.

For *SSCA2* eager conflict detection, multiple-writers STMs (STM^2 and TinySTM) perform considerably better than the other STMs. This seems to indicate that early detection of conflicts reduces the STM overhead for this application. *SSCA2* differs from the other benchmarks in that it shows a bursting and irregular behavior with

higher number of short, read-write transactions per second (high commit rate). Lazy conflict detection STMs (TL2 and NOrec) fail to acquire all required locks at commit time because other transactions commit in the meantime. Even if these commits do not generate actual conflicts, NOrec still needs to re-validate the elements in the read-sets. Increasing the number of concurrent threads also raises the probability that a transaction commits while another thread is validating its read-sets. The result is that commit time increases with the number of threads. Conversely, TinySTM and STM^2 acquire ownership of shared memory locations when a thread issues a write operation and maintain it throughout the execution of the short transaction, which proves to be a good choice for this particular case.

NOrec Unlike TinySTM, TL2 and STM^2, NOrec does not perform any bookkeeping. Runtime overhead is negligible and limited to the initialization and finalization of transactions. However, NOrec only allows one active writer transaction in the system at a time. We, thus, expect NOrec to perform better than STMs with bookkeeping when the level of contention is limited, but to gradually reduce performance when the number of writers per transaction increases (which depends on the application and the number of concurrent threads). Indeed, NOrec scales nicely for all applications except *Bayes* and *SSCA2*. On the other hand, bookkeeping allows STM^2 to support several concurrent writer transactions at a time. The result of combining concurrent writers and reduced runtime overhead is that STM^2 usually performs better or equal than NOrec. For applications with limited contention (*Genome*) or with a limited number of concurrent writers (*VacationLow* and *VacationHigh*), STM^2 and NOrec perform similarly. When the level of contention increases or there are several writers per transaction, STM^2 outperforms NOrec. This happens with *Bayes* (high contention) with N=32 threads (1.7x speedup) and with *SSCA2* (high number of concurrent writers) beyond N=4 threads (up to 6.4x speedup). In these cases, STM^2 keeps scaling up to 32 hardware threads, while rollbacks and re-validation limit NOrec's performance.

NOrec and STM^2 perform similarly for *Labyrinth*, which presents large transactions and a high conflict rate. For this kind of application, lazy conflict detection STMs usually have a disadvantage with respect to eager conflict detection STMs. NOrec, however, is able to make up for this disadvantage with its value-based conflict detection. The results is that NOrec introduces fewer false aborts than TL2.

STM^2 provides, on average, 2.1x speedup over NOrec (see Figure 5.6). Since NOrec has a low runtime overhead, our results prove that eager conflict detection and bookkeeping overhead is effectively absorbed by the auxiliary threads.

TL2 Both TL2 and STM^2 perform lazy data versioning, though TL2 detects conflicts at commit stage while STM^2 detects conflicts during a transaction's execution. Lazy conflict detection STMs introduce negligible validation runtime overhead but they may suffer from higher abort overhead, caused by the "wasted" time spent executing transactions that will abort, and high commit overhead (lock acquisition). Our experiments show that, indeed, TL2 performs well for applications with low contention, like *VacationLow* and *VacationHigh* (which perform mainly read operations) or *Genome* (limited contention). While STM^2 and TL2 are essentially equivalent for *Genome*, TL2 performs better than STM^2 when running *VacationLow* and *VacationHigh*. Applications with high contention or high commit rate, instead, pose challenges to TL2 due to frequent modifications of a centralized data structure [95]. For these kinds of applications, STM^2 outperforms TL2: with 32 hardware threads, STM^2 achieves 1.5x speedup over TL2 for *Bayes*, 1.6x speedup for *Labyrinth*, and 5.4x speedup for *SSCA2* (Figure 5.6). Note that, while STM^2 performs significantly better than TL2 for high contention applications, TL2 does not substantially outdistance STM^2 for applications with low-contention or applications with mostly-read transactions. The results show that, on average, STM^2 shows a 1.8x speedup over TL2.

TML STM^2 performs consistently and substantially better than TML for all STAMP benchmarks. While a global lock provides low runtime overhead and intrinsically guarantees serialization, performance is usually poor for applications with high contention and/or large transactions. Our experiments show that the serialization overhead induced by the use of a global lock with a high number of threads considerably reduces overall performance. As Figure 5.6 shows, STM^2 exceeds TML for applications with high contention, like *Bayes* (1.7x speedup) and *Labyrinth* (12.8x speedup), large readsets, like *VacationLow* (5.8x speedup), and read-write transactions, like *SSCA2* (4.6x speedup).

Summary Our results show that, on average, STM^2 outperforms all tested STMs. For applications with high contention (*Bayes* and *Labyrinth*) or bursting and irregular transactions with a high number of concurrent writers (*SSCA2*), STM^2 provides high speedups over lazy conflict detection STMs (up to 6.4x) or single global lock

STM (12.8x). For applications with low contention and mostly-read transactions (*VacationLow*, *VacationHigh* and *Genome*), STM^2 performs well with respect to lazy conflict detection and no bookkeeping STMs: Only TL2 outperforms STM^2 when running *VacationLow* and *VacationHigh*, while STM^2 still outperforms NOrec and TML for *VacationLow* and *VacationHigh* and NOrec, TML and TL2 for *Genome*. STM^2 provides the same performance, or even outperforms, lazy conflict detection and no-bookkeeping STMs for applications where lazy conflict detection provides advantages. Finally, STM^2 exceeds TinySTM with all the applications and provides speedups up to 12.3x over TinySTM for applications that are critical for eager conflict detection STMs, such as *VacationHigh*.

Our proposal largely overlaps computation and STM management operations and effectively reduces runtime overhead. STM^2 remarkably improves performance and provides the advantages of eager conflict detection STMs with the limited runtime overhead of lazy conflict detection STMs. Note that, given that all STMs run the same number of transactions and that STM^2 is faster than the other STMs (between 1.8x and 5.2x with 32 hardware threads, on average), it follows that STM^2's throughput (measured in number of transactions per second) is higher, despite the use of dedicated hardware threads to run STM operations.

5.6 Related Work

The use of extra threads to help the computation of main threads has been previously proposed, though for different goals. Auxiliary threads are usually employed to resolve unpredictable branches or cache misses that the main threads would have to stall upon otherwise [29, 33, 153] or to prefetch data from memory. Zilles et al. [166] explore using separate threads in a multithreading processor for exception handling to avoid squashing in-flight instructions.

Mehrara et al. [108] and Milovanovic et al. [113] propose the use of an auxiliary thread in lazy conflict detection STMs. Both proposals, however, use a centralized dedicated thread. Mehrara et al. [108] present STMlite, a software transactional memory that aims to automatically parallelize sequential applications. In this work, all the application threads send their memory modifications to the auxiliary thread, which, at commit time, serially performs the updates. This approach provides benefits when the

lock contention is high by serializing the memory updates in one thread. Milovanovic et al. [113] propose a combined OpenMP and STM runtime system based on an STM library, which performs lazy conflict detection and lazy versioning management. The authors introduce an additional separate thread for asynchronous eager conflict detection that aims to detect conflicts before the commit time and, therefore, reduce wasted time for doomed transactions. However, the authors did not implement an advanced synchronization mechanism between transactions and the associated dedicated thread. This unnecessarily forces the system at commit phase to repeat several checks already performed during the eager conflict detection phase. Both proposals suffer from a lack of scalability: the centralized auxiliary thread may become a bottleneck, especially for a high count of threads.

Casper et al. [25] use an FPGA connected to the AMD HyperTransport bus to accelerate conflict detection using bloom filters. Conflict detection is performed at commit phase by the accelerator and it is synchronous with the threads running on the normal cores which have to wait for the accelerator to complete conflict detection.

In contrast to previous work, STM^2 is a fully parallel STM: STM^2 assigns a dedicated auxiliary thread to each application thread for managing validation and bookkeeping involved in the main computation. These threads run on dedicated cores/hardware threads. Since each application thread has its own auxiliary thread for their transactional operations, unlike STMlite [108] and the approach proposed by Milovanovic et al. [113], we avoid having a single point of serialization. Finally, STM^2 and the work proposed by Casper et al. [25] are orthogonal: STM^2's auxiliary threads could be accelerated through dedicated hardware, such as FPGAs.

5.7 Conclusion

In conclusion, we have presented STM^2, a parallel STM system that offloads STM time-consuming management operations to auxiliary threads running on separate hardware threads. To the best of our knowledge, STM^2 is the first parallel STM that takes advantage of secondary hardware threads to accelerate STM functions and reduces overall overhead.

We tested STM^2 on an IBM POWER7 system, an aggressively multithreading processor designed for high performance. By overlapping computation and STM oper-

ations, STM^2 generally outperforms current, state-of-the-art STMs, namely TinySTM, TL2, NOrec and TML. Our experiments show average speedups between 1.8x and 5.2x over the tested STMs, with peaks up to 12.8x, with 32 hardware threads. We conclude that auxiliary threads effectively absorb the overhead of transactional bookkeeping and conflict detection, considerably improving the overall performance.

Chapter 6

Enhancing the Performance of Assisted Execution Runtime Systems through Hardware/Software Techniques

6.1 Introduction

As presented in the previous chapter, adopting assisted execution model for STMs reduces runtime overhead, therefore provides significant performance improvement. However, the main drawback of assisted execution models is the generally low processor utilization and the waste of resources, especially in phases when applications could use all available hardware threads. Waste of hardware resources cannot be tolerated in high-efficiency system (e.g., Exascale systems) and a tighter interaction between hardware and software is essential to reach high level of system efficiency [45].

This work explores the use of fine-grained hardware resource allocation to increase overall processor utilization and application's performance when a static partition of hardware resource in assisted execution runtime systems leads to sub-optimal performance and processor under-utilization. As opposed to coarse-grained resource allocation (adding or removing cores/hardware threads to a particular task) that can be

implemented at software level, fine-grained resource allocation (partitioning renaming registers, load/store queue entries, ROB slots, etc.) requires a collaboration between the software and the underlying hardware. Although fine-grained resource allocation requires a deep understanding of all the layers involved, from the hardware to the applications, it has the potential to provide higher performance and better adapt to frequent changes in the application's behavior.

In this chapter, we apply fine-grained hardware resource partitioning to STM^2 (*Software Transactional Memory for Simultaneous Multithreading* processors) [82]. In order to increase processor utilization and overall performance through fine-grained resource allocation, we used an integrated hardware/software approach where system functionalities are divided among three different components: 1) STM^2 is enriched with a runtime mechanism that automatically detects computing demand of application and auxiliary threads and drives the underlying hardware actuators; 2) the hardware enforces resource partitioning among the running threads; 3) the operating system provides an interface between STM^2 and the dynamic hardware resource allocation mechanism. This work spans the full hardware/software stack: we leverage the IBM POWER7 *hardware thread prioritization* [17, 20, 145, 162] to dynamically partition hardware resource (e.g., renaming registers or load/store queue entries) between application and auxiliary threads; we use a special version of Linux 2.6.33 patched to enable the full range of hardware priorities from within user level programs.

In this work, we begin by proposing a set of static techniques that can be applied when configuring STM^2 to partition hardware resources between application/auxiliary thread pairs. To this extent, we explore all possible configurations to apply fine-grained resource allocation to STM^2 and their performance implications. We show that static approaches work for straightforward optimizations but might not work for complex optimizations, such as resource partitioning within transactions. If the transaction structure is irregular or present bursts of transactional operations, a static approach may lead to sub-optimal performance or, in the worst case, performance degradation. Moreover, programmers need to manually explore all the possible settings and select the best performing configuration. Hence, we propose an adaptive solution that automatically partitions hardware resources between application and auxiliary threads at run time, transparently to the programmer and with no need of manual reconfiguration. Our adaptive solution monitors the computing power demand of application and

auxiliary threads and adapts to 1) phases within an application, 2) different behaviors of each application/auxiliary threads pair within the same application, and 3) the structure of the particular transaction executed by a thread at a given moment.

We test our proposals on a real IBM POWER7 system using two sets of benchmarks: first, we explain the potentialities of fine-grained resource allocation using Eigenbench [73] (a malleable TM micro-benchmark developed to explore TM systems' corner cases) and then we apply our solutions to STAMP applications [22]. Experimental results show that static approaches are only effective for simple scenarios while more realistic and complex applications require the use of adaptive solutions. Performance results for the adaptive solution show improvements that match the static approaches, for simple cases, and up to 65% and 85% over the standard STM^2 design for more complex scenarios where static approaches fail to provide optimal performance.

The rest of this chapter is organized as follows: Section 6.2 provides a short background on IBM POWER7 hardware thread priority mechanism. Section 6.3 details static techniques and the impact of POWER7 hardware thread prioritization on STM^2. Section 6.4 describes cases where static solutions fail to reduce load imbalance and introduces a new adaptive solution. Section 6.5 provides experimental results for Eigenbench and for applications from the STAMP benchmark suite. Section 6.6 details the related work. Finally, Section 6.7 concludes this work.

6.2 Hardware resource partitioning

Fine-grained hardware resource allocation generally requires hardware support to dynamically partition resources at run time with acceptable latency. A wide range of mechanisms to control hardware resources allocated to a particular core or hardware thread have been proposed in the literature [26, 30, 101, 102]. Some of these proposals have been implemented in real IBM [61, 145, 146] or Intel [76] processors, which allows system developers to implement fine-grained resource allocation solutions on real systems. In this work fine-grained hardware resource allocation for STM^2 is implemented upon IBM POWER7 processors, using the hardware thread prioritization mechanism to dynamically assign processor resources to the running threads at run time. This section briefly describes the POWER7 hardware prioritization mechanism.

Priority	**Priority level**	**Privilege level**	**or-nop inst.**
0	Thread shut off	Hypervisor	-
1	Very low	Supervisor	`or 31,31,31`
2	Low	User	`or 1,1,1`
3	Medium-Low	User	`or 6,6,6`
4	Medium	User	`or 2,2,2`
5	Medium-high	Supervisor	`or 5,5,5`
6	High	Supervisor	`or 3,3,3`
7	Very high	Hypervisor	`or 7,7,7`

Table 6.1: Hardware thread priority levels in the IBM POWER7 processor.

IBM POWER7 [4, 162] processors are out-of-order, 8-core design with each core having up to 4 SMT threads (32 hardware threads in total). Each core region (or “chiplet”) contains a 32KB 4-way set associative L1 I-cache and a 32KB 8-way set associative L1 D-cache, a private per-core 256KB L2 cache and a 4MB portion of the shared 32MB L3 cache. Each POWER7 core can run in single-thread (ST) mode, SMT2 (two threads executing on a core concurrently) or SMT4 (three or four threads executing on a core) mode. For capacity computing (i.e., multiple independent, serial jobs running in parallel), both SMT2 and SMT4 modes are expected to provide benefits. For capability computing (i.e., parallel applications with high degree of parallelism), SMT4 may not show extra benefits [4].

Besides multi-core and multithreading capabilities, IBM POWER7 processors provide a mechanism to partition hardware resource within a core by fetching and decoding more instructions from one hardware thread than from the others [145]. Each hardware thread in a core has a *hardware thread priority*, an integer value in the range of 0 (hardware thread off) to 7 (the core is running in single thread mode), as illustrated in Table 6.1. The amount of hardware resources assigned to a hardware thread is proportional to the difference between the thread’s priority and the priorities of the other hardware threads in the same core. In general, the higher the priority of a hardware thread, the higher the amount of hardware resources assigned to that thread (if the other hardware thread priorities are constant).

The priority value of a hardware thread in POWER7 processors can be controlled by software and dynamically modified during the execution of an application. IBM

POWER7 processors provide two different interfaces to change the priority of a thread: issuing an `or`-nop instruction or using the *Thread Status Register* (`TSR`). The current thread priority of a hardware thread can be read from the local `TSR` register using a `mfspr` instruction. As Table 6.1 shows, not all hardware thread priority values can be set by applications: user software can only set priority levels 2, 3, 4; the operating system (OS) can set 6 out of 8 levels, from 1 to 6; the Hypervisor can span the whole range of priorities. In order to use all possible levels of priorities, a special Linux 2.6.33 kernel patched with the Hardware Managed Threads priority (HMT) patch [17, 18, 19] is required. This custom kernel provides two interfaces (a `sysfs` and a system call) through which the users can set the current hardware thread priority, including the ones that require OS or Hypervisor privilege (the OS issues a special Hypervisor call to set priority 0 and 7). The system call interface (`hmt_set()`) is more suitable for the purpose of this work than the `sysfs` interface because it introduces lower overhead when frequently changing application and auxiliary thread priority.

6.3 Static Fine-Grained resource partitioning

Runtime systems benefit from the assisted execution model if application threads often require support to perform high-overhead operations [98, 109, 166]. Under these hypothesis, devoting hardware threads to perform runtime operations rather than main computation may provide higher performance than using all available hardware resources to run application threads. STM^2, in particular, provides significant speedups over canonical STM systems (between 1.8x and 5.2x on average) [82] for applications that spend a considerable amount of time performing transactions and the overhead of TM operations limits scalability. However, similarly to other assisted execution systems, STM^2 may not fully utilize the processor's resources. For example, an application could spend most of its execution time in an embarrassingly parallel phase (Section 6.3.1) or alternate accesses to shared locations with private variables within transactions (Sections 6.3.2.1 and 6.3.2.2).

This section describes fine-grained resource allocation techniques that can be applied to STM^2 at configuration time to improve processor utilization and efficiency

when auxiliary threads are mostly idle or overloaded.[1] Applying these techniques requires an comprehensive understanding of the IBM POWER7 hardware thread priority mechanism. Previous studies [17, 103, 111] focus on IBM POWER5 processor generations which, to the contrary of POWER7, feature only two hardware threads per core rather than four. In order to better explain the performance implications of the POWER7 hardware thread prioritization mechanism and the effects of fine-grained resource partitioning, this section follows a step-by-step approach. Experiments are performed using Eigenbench [73], a simple TM micro-benchmark that allows programmers to tune orthogonal TM characteristics, such as the number of local accesses outside or inside transactions, the conflict level, or the number of transactional operations per transaction. Eigenbench performs N consecutive iterations of a computation block, where each block consists of an embarrassingly parallel computation part and a transaction. We properly tune Eigenbench to create challenging scenarios for STM^2. We then leverage the IBM POWER7 hardware thread priority mechanism to improve STM^2's performance in these challenging scenarios. In the following sections, AT_p denotes the priority of an application thread, AxT_p the priority of an auxiliary thread, and $\Delta_p = AT_p - AxT_p$ the difference between the priority of an application thread and its corresponding auxiliary thread. Positive values of Δ_p denote that an application thread has higher priority than its paired auxiliary thread, whereas negative values of Δ_p indicate that the auxiliary thread has higher priority than its corresponding application thread.

6.3.1 Embarrassingly parallel phases

During embarrassingly parallel phases, threads perform computation on private data and do not need to protect accesses to memory locations. In STM^2, auxiliary threads paired with application threads not performing transactions at a given time sit idle, waiting for a new transaction to start. Since each thread runs on a dedicated hardware thread and the system is not over-provisioned, this design may lead to overall processor under-utilization. In fact, waiting auxiliary threads do not perform any useful work but consume hardware resources that could be used by application threads. A naïve solution

[1]No application's source modification is required in order to apply these techniques. Since STM^2 is transparent to applications (which are not aware of auxiliary threads), using these techniques from within the application would result in a very complicated task and requires compiler assistance.

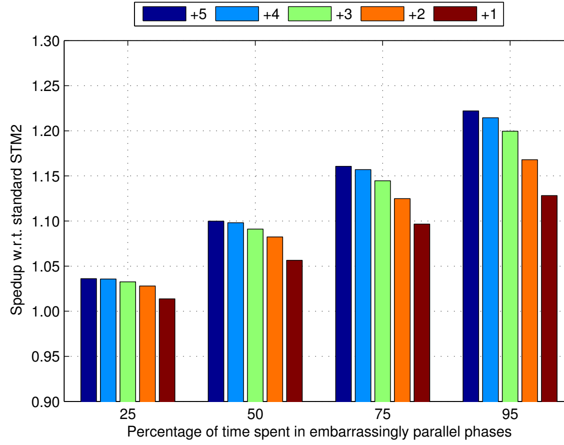


Figure 6.1: Performance impact of reducing the priority of auxiliary threads when varying the percentage of time spent performing embarrassingly parallel computation and the value of Δ_p. The graph shows that the performance improvement obtained by reducing the priority of the idle auxiliary threads is proportional to the percentage of time the application spends in embarrassingly parallel phases and to Δ_p. *naïve-EP* denotes suspending/resume in embarrassingly parallel (EP) phases.

to this problem consists of suspending waiting auxiliary threads and resuming their execution as soon as their paired application threads enter a transaction. This approach usually reduces responsiveness, which may limit overall performance, especially if the application frequently alternates short transactions and embarrassingly parallel phases. Moreover, suspending idle auxiliary threads, in general, does not increase processor utilization: the hardware thread that was running the auxiliary thread is released back to the operating system (OS) which may decide to either run another task or leave it idle. If there is no other runnable task available to the idle hardware thread, the OS may reduce the priority of the idle hardware thread, implicitly increasing the performance of the other hardware thread. This decision is not under the control of STM^2 and depends on the system status at the time of suspending an auxiliary thread. On the other hand, spinning usually guarantees higher responsiveness, which lead us design STM^2 with this approach, at the cost of unnecessarily consuming hardware resources without making any progress. In order to increase processor utilization in embarrassingly phases while maintaining high responsiveness, we reduce the hardware priority of spinning auxiliary threads (AxT_p) and restore it to its initial value as soon

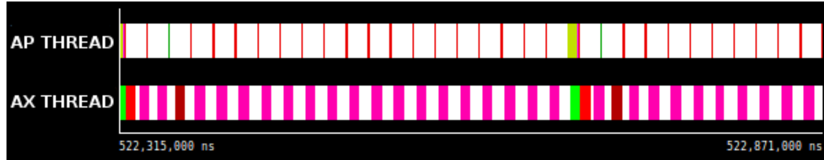
as the corresponding application threads start a new transaction.

The impact of reducing auxiliary threads priority during embarrassingly parallel computation phases on the performance of the whole application depends on the percentage of time the application spends performing embarrassingly parallel computation and the amount of extra hardware resources assigned to application threads (Δ_p). Figure 6.1 shows the performance improvement of Eigenbench over STM^2 when running 1000 iterations per thread, with one transaction per iteration. In this experiment, the number of transactional operations per iteration is fixed to 20.[1] We then vary the percentage of time spent by Eigenbench in embarrassingly parallel computation phases from 25% to 95% and the value of the hardware thread priority of the auxiliary threads (AxT_p) while keeping $AT_p = 6$. This experiment only focuses on the performance improvement obtained from reducing the hardware thread priority of auxiliary threads during embarrassing parallel phases, hence $AT_p = AxT_p = 6$ inside transactions. As expected, reducing AxT_p during embarrassingly parallel computation phases provides performance improvements proportional to the percentage of time the application spends in embarrassingly parallel computation. Figure 6.1 also shows that the best performance values are obtained with $\Delta_p = 5$ (i.e., $AT_p = 6$ and $AxT_p = 1$). This is an important design point because this value of Δ_p can only be achieved through the HMT Linux patch. Had we limited the use of priority to the user-available levels, the maximum Δ_p would have been 2 ($AT_p = 4$ and $AxT_p = 2$), which would provide performance improvement of 16.8% (in the 95% case) instead of 22.3% obtained with $\Delta_p = 5$. This performance improvement comes essentially free of any drawbacks, as reducing hardware resources does not have any impact on the performance of waiting auxiliary threads.

The graph also reports the performance improvement obtained suspending idle auxiliary threads (*naïve-EP*) and resuming them as new transactions begin. Suspending waiting auxiliary threads provides some performance improvement, mainly because the system only runs one application and, thus, there is a high probability that the OS will reduce the hardware thread priority of the hardware thread previously running the waiting auxiliary thread. However, the performance improvement achieved with this approach does not match the one obtained with $\Delta_p = 5$. This is due to two main

[1] Here, and in the rest of the chapter, Eigenbench is configured to perform 10% of transactional writes.

(a) Standard case.



(b) Static allocation of extra hardware resources to application threads.

Figure 6.2: Frequently idle auxiliary threads within a transaction. (a) In this scenario, application threads issue transactional operations at a low rate, thus, auxiliary threads are frequently idle. In this trace white denotes local computation within a transaction while colored bars denote transactional reads or writes. (b) Application threads receive more hardware resources ($AxT_p = 1$ and $AT_p = 6$) but auxiliary threads are still able to complete all TM operations before the corresponding application threads reach the commit phase, hence, the transaction's total execution time is reduced. The elapsed time in both traces is the same.

reasons: first, the OS is free to schedule any other process or kernel daemon on idle hardware threads previously occupied by the waiting auxiliary threads. This external process may introduce even larger slowdown on the applications threads (data cache lines eviction, TLB entries eviction, resource contention). Second, the overhead of resuming auxiliary threads may reduce the overall benefit.

6.3.2 Load imbalance inside transactions

Load imbalance [18, 19, 139, 160] is a well-known problem for parallel applications that need to synchronize at determined points, such as at barriers or fork/join constructs. Load imbalance happens when one or more threads in a parallel application have more work to perform than the others, with the result that the whole application proceeds at the speed of the slowest threads, which may severely limit overall performance.

In STM^2, each application/auxiliary thread pair needs to synchronize at the end of each transaction (`commit()`) before moving to the next phase. For each application/auxiliary thread pair, load imbalance may occur because: 1) the application

thread issues TM operations at a low rate, thus its corresponding auxiliary thread is frequently idle (Section 6.3.2.1), and 2) the auxiliary thread has not completed all TM management operations when the corresponding application thread reaches the commit phase (Section 6.3.2.2). The next sub-sections explain these scenarios with details.

6.3.2.1 Overloaded application threads

Figure 6.2 shows a partial execution trace (one transaction) of a scenario in which application threads perform TM operations at a low rate. In this figure, local computation (operations on private variables) is depicted in white and TM operations are drawn as colored bars. In order to obtain these execution traces, we instrumented STM^2 and produced traces that can be visualized with Paraver [130], a performance analysis tool commonly used to study parallel applications. In this experiment Eigenbench is configured in such a way that application threads perform N_{local} local operations for every shared access (N_{shared}). In the example shown in Figure 6.2, $N_{local} = 300$ and $N_{shared} = 20$ (the total number of operations is $N_{local} \times N_{shared} + N_{shared} = 6,020$), which results in the auxiliary thread being idle for 95% of the time during the execution of a transaction.

Figure 6.2a shows the standard STM^2 case: the auxiliary thread is frequently idle but consumes hardware resources by spinning on the communication channel for incoming read/write messages. The application thread, on the other hand, can only use a partial amount of the shared hardware resources, with the results that its speed is limited. In this simple scenario the programmer could configure STM^2 to reduce the priority of the auxiliary thread (AxT_p), therefore assigning more hardware resources to the application thread. Figure 6.2b shows the effect of setting $AT_p = 6$ and $AxT_p = 1$ ($\Delta_p = 5$): performance improves considerably by reducing AxT_p and assigning extra hardware resources to the application thread. In fact, although the auxiliary thread proceeds at slower speed than the one in Figure 6.2a (the trace shows that each TM operation now takes longer), the application thread does not have to wait and can proceed with its computation. This happens because the auxiliary thread has still enough time to complete all TM operations before its corresponding application thread reaches the commit phase, thus the application thread does not wait to complete the transaction.

Unfortunately, setting the correct values of AT_p and AxT_p is not always straightforward: since the internal design of the IBM POWER7 hardware thread priority mecha-

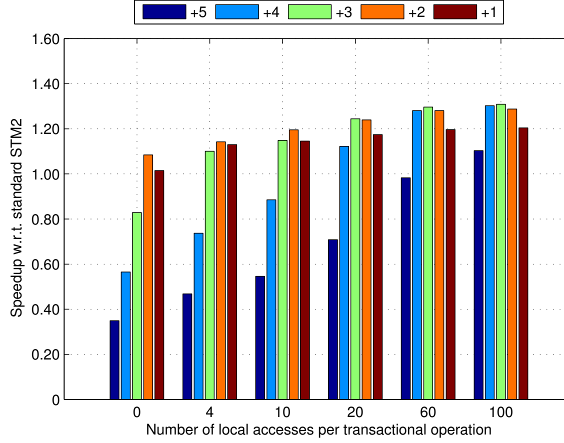


Figure 6.3: Performance impact of reducing the priority of auxiliary threads in presence of load imbalance within transactions (overloaded application threads). In this graph, we vary the number of local accesses per transactional operation (N_{local}) and the value of Δ_p. The results show that the best value of Δ_p is not always the same and that aggressive values of Δ_p provide performance improvement only when N_{local} is large.

nism is not symmetric [17], the performance degradation of the lower priority thread is usually higher than the performance improvement of the higher priority thread. This design does not lead to performance degradation when reducing the priority of auxiliary threads that are actually not doing any progress, like in embarrassingly parallel computation phases. However, applying the wrong set of priorities when both threads are performing useful work may reverse the imbalance, with the final effect of worsening the overall performance.

In order to quantify the effect of fine-grained resource allocation on applications with imbalanced transactions, we performed a complete design space exploration, varying the number of accesses to local variables (N_{local}) per TM operation (N_{shared}) within a transaction and the priority values of the auxiliary threads (AxT_p); $AT_p = 6$ in all cases. The result of this design space exploration is reported in Figure 6.3. When the number of local accesses is limited or null, excessively reducing AxT_p reverses the imbalance: auxiliary threads become the bottleneck and application threads have to wait at commit phase for their auxiliary threads to complete their work. This often leads to performance degradation, especially when the priority difference is large (e.g.,

(a) Standard case.

(b) Spin-only: Static allocation of extra resources to auxiliary threads at commit phase.

(c) Entire transaction: Static allocation of extra resources to auxiliary threads throughout the whole transaction.

Figure 6.4: Overloaded auxiliary threads. In the traces, white denotes transaction computation (both local and shared accesses) while light green denotes application threads' waiting time at commit phase. In this scenario auxiliary threads are overloaded and cannot complete all TM operations before their corresponding application threads reach the commit phase. Increasing the amount of hardware resources assigned to auxiliary threads improves overall performance. The elapsed time is the same for all the traces.

$\Delta_p \geq 4$). For $N_{local} = 0$, reducing the hardware thread priority of auxiliary threads degrades performance up to 63% ($AT_p = 6$ and $AxT_p = 1$, $\Delta_p = 5$). As the number of local accesses per TM operation increases, auxiliary threads are able to complete their work even with fewer hardware resources: for $N_{local} = 100$, aggressive settings achieve overall performance improvement of 44%.

6.3.2.2 Overloaded auxiliary threads

Some TM management operations, such as read-set validation or conflict detection, require a variable amount of time to be completed. For example, the read-set validation

overhead depends on the number of individual shared memory locations read during a transaction and the number of concurrent writers. The former determines the size of the read-set while the latter determines the frequency with which read-set validation is performed.

STM^2 is an eager-conflict detection STM, thus read-set validation is performed when a potential conflict arises. Note that, although required, not all read-set validations result in aborting the transaction. If an application triggers several read-set validations, auxiliary threads may not be able to complete all their TM operations before the corresponding application threads reach the commit phase. If such a situation arises, application threads are forced to wait at commit phase. Figure 6.4a illustrates this case: the auxiliary thread is not able to complete all TM management operations before its corresponding application thread reaches the commit phase, thus the application thread is forced to wait, effectively serializing part of computation and TM management operations. In the trace, application thread's waiting time at commit phase is denoted with light green while white depicts the execution of a transaction (both local computation and transactional operations).

Eigenbench does not allow the user to control the number of read-set validations per transaction, thus, in order to create the scenario in Figure 6.4a, we introduced extra (although not always necessary) read-set validations to simulate potential conflicts induced by large read-sets and large numbers of concurrent threads. In scenarios such as the one depicted in Figure 6.4a, prioritizing auxiliary threads ($\Delta_p < 0$) may provide performance benefits. This technique can be applied just at commit phase (Spin-only) or throughout the whole transaction (Entire Transaction).

Spin-only: Figures 6.4b shows how reducing AT_p while an application thread is waiting at commit phase speeds up the execution of TM management operations, achieving overall performance improvement. This solution, similarly to the case described in Section 6.3.1, is straightforward and does not introduce any performance degradation because application threads do not perform useful work while waiting at commit phase. In particular, the figure shows the case in which $AT_p = 1$ and $AxT_p = 6$ ($\Delta_p = -5$). Comparing Figures 6.4a and 6.4b, there is no performance degradation for the application thread computing phase (white in the traces), while the spinning time (light green) is considerably reduced. Performance improvement, in this case, is proportional to the spinning time reduction. Similarly to the the case described in

Section 6.3.1 (see Figure 6.1), overall performance improves with the decrease of AT_p, thus the best performance is achieved with $\Delta_p = -5$.

Entire transaction: Figure 6.4c shows a solution that decreases the priority of the application thread at the beginning of the transaction and maintains $\Delta_p < 0$ for the entire transaction execution. This approach is more aggressive than the previous spin-only solution: the performance of application threads considerably reduces by prioritizing auxiliary threads during the transaction computation phase. This can be observed by comparing Figures 6.4a and 6.4c: the application thread computing phase (white) takes considerably longer than in the standard case. On the other hand, the auxiliary thread does not accumulate too many pending TM operations, hence its corresponding application thread has to spin for less time at commit phase. The net result is that, with the more aggressive approach, the performance improves with respect to both the baseline (65%) and the safe, spin-only approach (7%). Note that statically reducing the priority of application threads also has the side effect of reducing the rate at which application threads inject messages into the communication channel. Consequently, auxiliary threads might spend time waiting for the next incoming message, which would reduce the net benefit. This situation may arise especially for large negative values of Δ_p but we have not measured any slowdown in our experiments.

As discussed above, the number of read-set validations per TM operation depends on application characteristics, such as the number of concurrent writers. Figure 6.5 shows the impact of statically increasing AxT_p ($\Delta_p < 0$) at the beginning of the transaction when the number of read-set validations per TM operation decreases. The graph also shows the performance of reducing AT_p to one ($\Delta_p = -5$) when application threads wait at commit phase (i.e., the case depicted in Figure 6.4b). Finally, the graph reports the effect of suspending waiting applications threads and resuming them once the transaction is ready to commit (*naïve-spin*). The figure reports the performance improvement over the standard STM^2 when performing read-set validation every N transactional operations (1:N) and varying the value of Δ_p. These experiments show that increasing AxT_p for transactions that require a high number of validations generally provides higher performance improvements than just reducing AT_p at commit phase or suspending/resuming waiting applications. For example, when performing one validation for every transactional operation (1:1), increasing AxT_p from the beginning of the transaction provides performance improvement of 65% over the standard STM^2

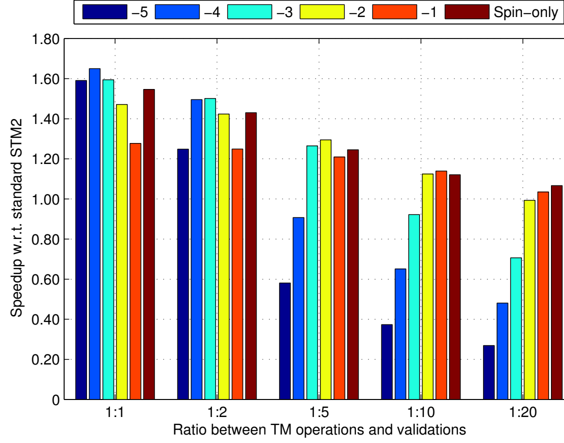
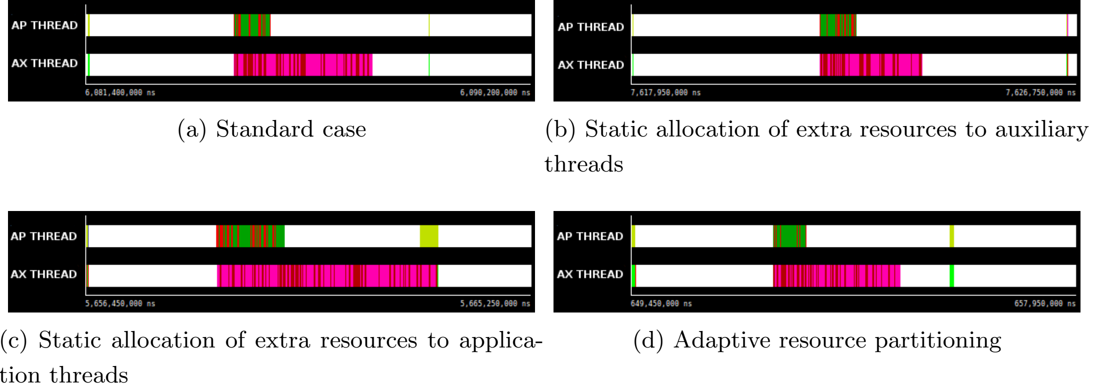


Figure 6.5: Performance impact of increasing the priority of overloaded auxiliary threads when varying the number of read-set validations per transactional operation and Δ_p. The graph shows that the right value of Δ_p is not always the same and that aggressive settings of Δ_p are only suitable when the ratio read-set validations to transactional operation is high.

while reducing AT_p to one at commit phase provides 55% performance improvement and suspending/resuming waiting application threads provides 40% performance improvement. On the other hand, reducing AT_p when an application thread is spinning at commit phase is a safe operation that does not introduce any measurable performance degradation. This technique can, therefore, be applied as fall-back mechanism in case a perfect balance between application and auxiliary threads cannot be achieved by increasing AxT_p at the beginning of a transaction.

Finally, the best value of Δ_p is not always the same for all ratios and aggressive settings are only possible when the ratio number of read-set validations to transactional operations is high. Figure 6.5 shows, in fact, that incorrect settings of AT_p and AxT_p when prioritizing auxiliary threads may lead to considerable performance degradation (up to 70%), especially if the number of read-set validations per transaction is low. As for the case of reducing AxT_p for frequently idle auxiliary threads (Section 6.3.2.1), manually setting AT_p and AxT_p is a complicated task, even for simple micro-benchmarks.



(a) Standard case

(b) Static allocation of extra resources to auxiliary threads

(c) Static allocation of extra resources to application threads

(d) Adaptive resource partitioning

Figure 6.6: Irregular transactions with bursts of transactional operations. In this example Eigenbench executes a burst of transactional operations in the middle of the transaction. White denotes local computation within transaction, colored bars denote transactional reads and writes, the light green at the end of the transaction denotes application threads waiting at commit phase. Static approaches provide sub-optimal performance because decreasing/increasing AxT_p improves performance in one phase but suffers slowdown in the other. The adaptive solution, instead, properly adapts to the transaction structure and partition hardware resources on demand.

6.4 Adaptive Fine-Grained resource partitioning

Section 6.3 shows that, for simple scenarios, setting the best values of AT_p and AxT_p can be done at configuration time by an expert programmer. For example, setting $AT_p = 6$ and $AxT_p = 1$ provides considerable performance improvements for embarrassingly parallel computation phases (Section 6.3.1), proportional to the percentage of time spent by the application in those phases (see Figure 6.1). In other cases, instead, setting the right value of Δ_p is not trivial and depends on the actual work performed by application and auxiliary threads. In certain cases (Figure 6.3), the right decision is to reduce AxT_p and give more hardware resources to the application thread; in others (Figure 6.5), the auxiliary thread is the bottleneck and increasing its priority improves overall performance. In both cases, the right value of Δ_p depends on the structure of the transaction (i.e., uniform versus burst structure, ratio of shared and local accesses, etc.). If the structure of the transaction is not uniform, depends on input parameters or changes during the execution of the applications, statically setting the proper values of AT_p and AxT_p becomes a daunting task and may result in performance degradation.

Figure 6.6a shows an execution trace of a transaction with a burst of accesses to shared memory locations roughly starting in the middle of the transaction: the application thread performs some local computation (white in the trace) followed by a burst of shared memory accesses (the colored bars in the trace denote transactional read and write operations), and then performs computation on local data.[1] This simple example shows one of the reasons why the original STM^2 design provides performance improvement over other state-of-the-art STM systems: a large part of the execution of TM operations overlaps with the application computation. At the same time, the auxiliary thread is able to complete all TM operations before the application thread reaches the commit phase, thus, application threads run almost as if they were oblivious of the STM runtime library. However, the same trace shows that the auxiliary thread is mainly idle during the local computation phases and considerably overloaded when the sudden burst of shared memory accesses starts. As with many static approaches, the main problem in this example is that it is not trivial to configure, at compile time, the values of AT_p and AxT_p that provide the best performance for both local computation and bursts of shared memory accesses.

Noticing that the burst of shared accesses results in a considerable amount of work for the auxiliary thread, one possible approach is to increase AxT_p ($\Delta_p < 0$): Figure 6.6b shows the effect of setting $AT_p = 5$ and $AxT_p = 6$ ($\Delta_p = -1$). In this case, although the performance of the auxiliary thread improves considerably during the execution of the TM operations, the performance degradation suffered in the local computation phases outweighs the improvement, which results in an overall 27% slowdown with respect to the standard STM^2 design.

The opposite approach consists of reducing the priority of the auxiliary thread to benefit the application thread. Figure 6.6c shows that the performance of the application thread during the local computation phase improves by 12%, while, obviously, the performance of the auxiliary thread when performing TM operations decreases. In fact, the trace shows that the auxiliary thread is still performing TM operations when the application thread reaches the end of the transaction, thus, the application thread has to wait at commit phase. Although local computation phases are predominant in

[1] In this experiment, we use a modified version of Eigenbench. By design, Eigenbench issues TM operation uniformly, i.e., one TM operation every N_{local} local operations, thus the standard version of Eigenbench does not allow us to generate bursts of TM operations.

this transaction structure and part of the TM operations overlaps with the second local computation phase, there is still a slight performance degradation. Neither solution is optimal, as both of them gain and suffer in opposite phases during the execution of the transaction. The common problem to both solutions is that static approaches seldom adapt to non-uniform structures, such as the one presented in Figure 6.6a.

This section introduces an automatic solution that adapts, at run time, to the transaction's structure and automatically sets the best values of AT_p and AxT_p. This adaptive solution is based on heuristics and on the lessons learned when applying the static solutions described in Section 6.3. The proposed heuristics are designed according to the following key principles:

P1 Reducing the hardware thread priority of either application or auxiliary threads introduces an asymmetric performance degradation [17]. Thus, we need to be careful when decreasing the priority of a thread, especially for large values of $|\Delta_p|$.

P2 Reducing the priority of a waiting auxiliary thread considerably improves performance (up to 44%), as shown in Figures 6.1 and 6.3, without any performance degradation. The heuristics should decrease AxT_p whenever the application enters a (large) embarrassingly parallel section.

P3 If application threads issue bursts of TM operations, their corresponding auxiliary threads may not be able to complete all TM operations before the application threads reach commit phase. The heuristics should consider prioritizing auxiliary threads ($\Delta_p < 0$) in such scenarios. Figures 6.5 and 6.6b show that increasing AxT_p directly affects application threads' performance, thus auxiliary thread prioritization must be done judiciously.

P4 As proved in Section 6.3, in some cases, large values of Δ_p provide higher performance improvements. However, this required the use of the system call `hmt_set()`, which introduces some overhead. We tend to modify the priority of waiting threads (mainly the auxiliary threads), as this directly affects the value of $\Delta_p = AT_p - AxT_p$.

Besides these basic key principles, the adaptive solution employs the static mechanisms that do not depend on the actual application and auxiliary thread structure,

such as reducing AxT_p during embarrassingly parallel phases or reducing AT_p when spinning at commit phase. For short embarrassingly parallel sections or for cases in which all TM operations are completed when an application thread reaches the commit phase, the overhead of invoking a system call may outweigh the performance improvement. To avoid such situations, the adaptive solution snoozes for a short time before actually issuing the system call. At the beginning of a transaction the initial values are $AT_p = AT_p(0) = 5$ and $AxT_p = AxT_p(0) = 5$. Since AT_p remains constant during the execution of a transaction ($P4$), these settings allow the enriched STM^2 to reach large positive values of Δ_p ($P2$) but also to be able to prioritize auxiliary threads ($P3$), depending on the structure of the transaction. With these initial settings, positive and negative values of Δ_p can be achieved by changing only the value of AxT_p, without the application threads issuing any `hmt_set()` ($P4$), hence reducing the run time overhead.

Detecting load imbalance: The first problem towards the development of an adaptive solution is how to determine, at run time, if there is load imbalance and which thread is the bottleneck. In order to detect load imbalance, we monitor the number of messages queued in the communication channel between application and auxiliary threads. In fact, if the application thread issues transactional operations at a low rate, the queue would be frequently empty. Conversely, if the application thread issues TM operations at a rate higher than the rate at which the auxiliary thread completes its work, the queue would gradually fill up. By monitoring the queue between application and auxiliary thread, we are able to effectively detect load imbalance with negligible overhead.

Reducing the priority of auxiliary threads: In case it is determined that an auxiliary thread is often idle (i.e., the queue is empty most of the time), the heuristic aims at reducing AxT_p so that the paired application thread receives more hardware resources. In order not to reverse the load imbalance, the heuristic decreases AxT_p only after a certain amount of consecutive attempts to dequeue a message that reveal that the queue is empty. This quantity is denoted as *snooze_time*. A constant *snooze_time* would decrease the priority regardless of the current value of AxT_p, however, according to principle $P1$ and previous work [17, 18, 19, 57, 107], the priority of a thread should be decreased judiciously, especially if the priority difference Δ_p is already large. On the other hand, Figure 6.3 shows that, even for small values of N_{local}, decreasing AxT_p provides benefits. The solution adopted here is to make *snooze_time* variable: every time

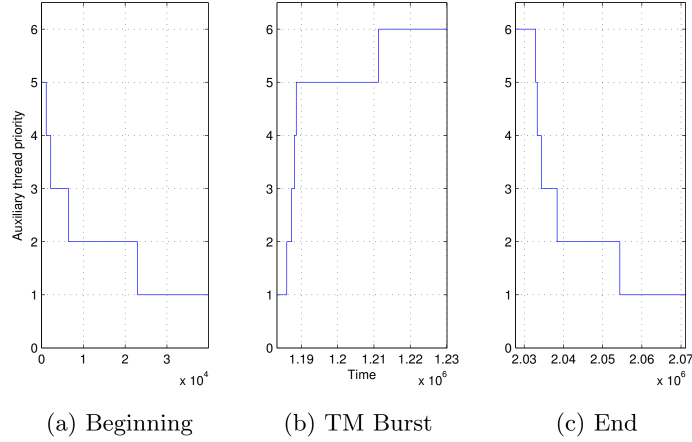
the auxiliary thread decreases its priority, the heuristic computes a new *snooze_time* value as a function of Δ_p. The general idea is that the larger the value of Δ_p, the larger the value of *snooze_time*, i.e., the value of AxT_p is reduced less often if the Δ_p is large. *snooze_time* is computed as:

$$snooze_time = a + b * f(\Delta_p)$$

where $f(\Delta_p)$ is a monotonically increasing function for $\Delta_p = 0...4$, and a and b are constants. *snooze_time* is small for small values of Δ_p, which allows the heuristic to quickly reduce AxT_p when the auxiliary thread is idle. As Δ_p increases, *snooze_time* increases as well, which makes the next priority change occur less often. The right $f(\Delta_p)$ function depends on the specific hardware resource partitioning scheme: For POWER7 systems, the performance degradation of reducing the hardware thread priority is exponential ($P1$), hence an $f(\Delta_p)$ expressed as an exponential function should confidently model performance. For $\Delta_p = 0...4$, however, an exponential function can be approximated with a less expensive polynomial: empirical tests show that $f(\Delta_p) = (\Delta_p + 1)^3$ quickly reacts for small values of Δ_p and progressively increases *snooze_time* for larger values of Δ_p. The values of $a = 40$ and $b = 10$ are also empirically determined, so the final function to compute the next value of *snooze_time* is the following:

$$snooze_time = 40 + 10 * (\Delta_p + 1)^3$$

Increasing the priority of auxiliary threads: In this adaptive system, the value of AxT_p at time t may be lower than its initial value $AxT_p(0)$. This may happen because the auxiliary thread was frequently idle before t and the heuristic decreased its hardware thread priority. If a burst of TM operations such as the one depicted in Figure 6.6a arises, the heuristic should increase the value of AxT_p or else the auxiliary thread will not be able to complete its work before its corresponding application thread reaches the commit phase (see Figure 6.6c). Raising the value of AxT_p should not be too impulsive as, if the burst is particularly short, it might be worth keeping $AxT_p < AxT_p(0)$ (Figure 6.3). The heuristic increases the value of AxT_p until it reaches $AxT_p(0)$ if there are more than `QS_THRESHOLD` elements in the queue (in the current implementation this value is 20), which denotes that the auxiliary thread is potentially accumulating work. As reported in Section 6.3.2.2, increasing the value of AxT_p so that $\Delta_p < 0$ reduces the rate at which application threads inject messages into



(a) Beginning (b) TM Burst (c) End

Figure 6.7: The adaptive solution automatically changes the value of AxT_p according to the structure of the transaction and the computing demand of application and auxiliary threads. These graphs show the values of AxT_p during one transaction for the case discussed in Figure 6.6d. The x-axis reports time since the beginning of the transaction.

the communication channel, which may reduce the overall performance. Nevertheless, there are critical cases, such as a burst of TM operations at the end of a transaction, in which this approach may increase the performance: auxiliary threads are allowed to increase their priority up to $AxT_p = 6$ (i.e., $\Delta_p = -1$) if the number of pending messages in the queue is larger than `QS_THRESHOLD_CRITICAL` (128 in our implementation). Notice that higher values of $|\Delta_p|$ (i.e., $\Delta_p = -2, -3, ..$) are also possible but cumbersome. Instead, in case the load cannot be balanced with $\Delta_p = -1$, the heuristic reduces the priority of the application thread while spinning at commit phase (see Figure 6.4b).

Figure 6.6d shows that the dynamic solution is able to adapt to the non-uniform structure of the transaction and provides higher performance than both static approaches. First, the adaptive solution reduces the priority of the auxiliary thread during the initial local computation phase, therefore improving the performance of the application thread. Next, when the sudden burst of accesses to shared memory locations starts, the adaptive solution gradually increases AxT_p and, if necessary, reaches values higher than AT_p ($\Delta_p < 0$). Finally, once the auxiliary thread has completed all its TM operations, the adaptive mechanism reduces AxT_p again, improving the performance of

the application thread in the last part of the transaction. The overall result is performance improvement of 15%, where static approaches result in performance degradation when decreasing or increasing AxT_p, respectively (based on the best values among all possible settings of AT_p and AxT_p).

The adaptive solution's heuristics can be evaluated with two different metrics: 1) the convergence to the best value of Δ_p, and 2) the speed at which the heuristics reach that value. Figure 6.7 shows the value of AxT_p (AT_p=$AT_p(0)$ throughout the execution) as function of the elapsed time since the beginning of the transaction, for a transaction with a burst of TM operations in the middle (the same case reported in Figure 6.6d). As Figure 6.7a shows, the adaptive solution successfully converges to $AxT_p = 1$ (stable state), first quickly and then, as Δ_p increases, more slowly (the steps get larger as Δ_p increases). When the sudden burst of TM operations starts (Figure 6.7b), the heuristic quickly adapts to the new scenario and converges to $AxT_p = 5$. Since increasing AxT_p does not depend on Δ_p, the steps are much smaller than in Figure 6.7a. In the meanwhile, the auxiliary thread has accumulated a considerable amount of work, thus, the heuristic further increases $AxT_p = 6$ ($\Delta_p = -1$), giving more hardware resources to the auxiliary thread (Figure 6.7b). Finally, once all the accumulated work has been processed, the heuristic automatically reduces the auxiliary thread priority until it reaches the value $AxT_p = 1$ (Figure 6.7c), similarly to what happens in the first part.

6.5 Experimental results

The evaluation of the adaptive solution presented in the previous section is performed on a IBM POWER7 system (8 cores, 4 hardware threads per core) equipped with 64 GB of RAM. STM^2 and all the applications are compiled with GCC 4.3.4 with optimization level `O3` and the results reported for each application are the average of 25 runs. In order to use all hardware priority levels, all tests are performed on a custom version of the Linux 2.6.33 kernel patched with the HMT patch [18]. In all the experiments, the Eigenbench and STAMP applications use all the available hardware threads (32 in the tested configuration): 16 application threads and 16 auxiliary threads. STAMP applications use the reference (large) input sets [22].

6.5.1 Eigenbench

Figure 6.6 shows the effect of statically increasing (Figure 6.6b) and decreasing (Figure 6.6c) AxT_p for transactions with non-uniform structure. In particular, Figure 6.6 depicts an example of a transaction with a burst of accesses to shared memory locations roughly starting in the middle of the transaction. For this particular example, neither statically decreasing nor increasing AxT_p provides performance improvement over the standard STM^2, while the adaptive solution effectively allocates hardware resources on demand, reaching overall performance improvements up to 15%.

An interesting observation is that, for transactions with burst of TM operations, the performance improvement depends on both the size and the position of the burst. The size of the burst clearly affects whether prioritizing application/auxiliary threads provides higher performance. In the extreme cases (bursts of 0% or 100%), the examples degenerate to the cases presented in Section 6.3 (Section 6.3.2.1 and 6.3.2.2, respectively). If a short burst of TM operations occurs at the beginning of a transaction, the auxiliary thread has enough time to complete all TM management operations before the application thread reaches the commit phase, even without applying fine-grained resource allocation. As the burst of TM operations moves towards the end of the transaction, the auxiliary thread may not be able to complete all the TM operations before its corresponding application thread reaches the commit phase. If that happens, the application thread will spin at commit phase, which leads to sub-optimal performance. Obviously, the worst case occurs when the burst of TM operations appears at the end of the transaction: in this case application and auxiliary threads essentially run sequentially, invalidating most of the advantages of assisted execution.

Figure 6.8 shows the performance of the best combination for the proposed static approaches (both prioritizing application and auxiliary thread) and the adaptive solution over the standard STM^2 design, when varying the size of the burst (from 20% to 75% of the transaction execution time) and the position of the burst (middle and end of the transaction). The experiments in the graph show that statically partitioning hardware resources provides performance improvement only for extreme cases (20% and 75%), either because there is a large part of the transaction in which the auxiliary thread is mainly idle, or because the burst is large enough to cause the application thread to spin at commit phase for a considerable amount of time. In the other scenarios (30%

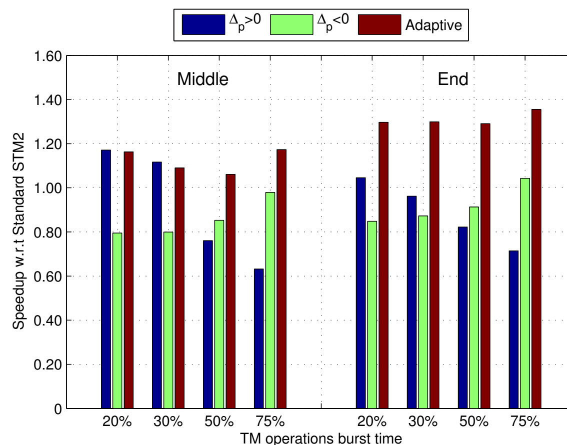


Figure 6.8: Performance of static (best values among all combinations) and adaptive solutions for application with not-uniform transaction structure and varying size/position of burst of shared accesses. The graph shows that the adaptive solution matches or outperforms static approaches and always provides performance improvement over the standard STM^2. EP denotes embarrassingly parallel phases.

and 50%), STM^2 effectively runs TM operations and computation in parallel. As Figure 6.6 shows, neither increasing nor decreasing AxT_p provides the best performance and both approaches incur performance degradation. The dynamic approach, on the other hand, 1) provides performance improvement over both static approaches and 2) more importantly, always outperforms the standard STM^2 for both the "Middle" and the "End" cases. This experiment shows that the automatic solution is able to adapt to the structure of the transaction, properly increasing or decreasing the value of AxT_p on demand. In a nutshell, the adaptive solution provides performance improvements between 6% and 38% over the standard $STM^2$and outperforms the best performance provided by both static techniques.

6.5.2 STAMP applications

This section shows that the cases examined in the previous section with Eigenbench are indeed common to more complex applications and pose challenges to assisted execution runtime systems, such as STM^2. To evaluate the proposed adaptive solution, we use applications from the STAMP benchmark suite, a set of applications widely used to

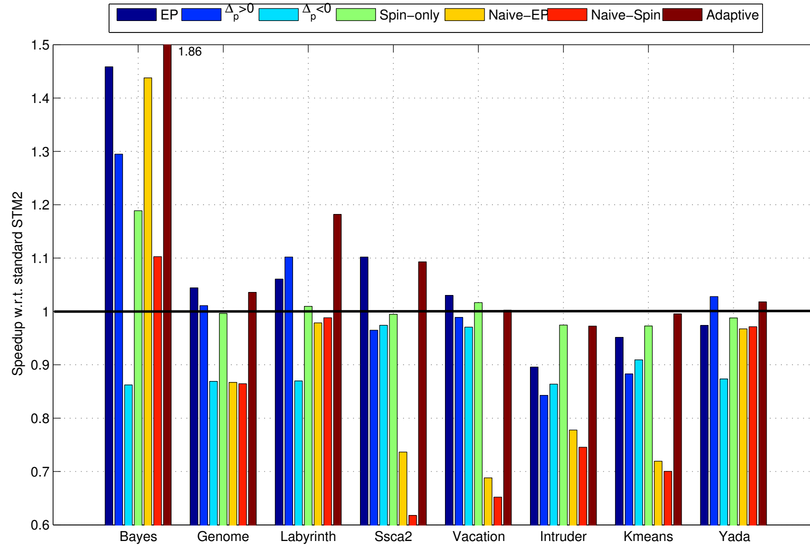


Figure 6.9: Performance impact of static (best values among all combinations) and adaptive solutions for STAMP applications. The adaptive solution matches or outperforms the static approaches for all applications (except *Vacation*).

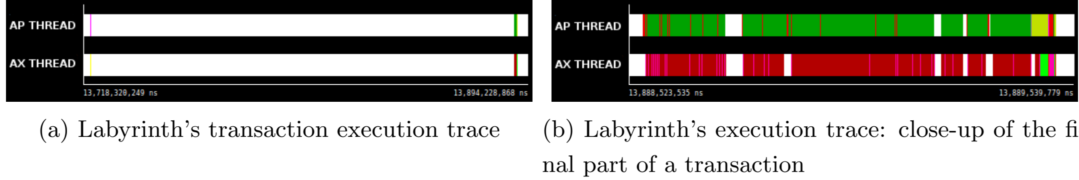
test transactional memory systems.

Figure 6.9 shows the performance of (separately) applying the static approaches described the Section 6.3 and the adaptive solution presented in the previous section. The graph reports, for each static technique, the best values of the pair (AT_p, AxT_p) among all possible configurations. Applications in Figures 6.9 can be divided into two groups: applications in the first group (*Bayes*, *Genome*, *Yada*, *Labyrinth*, and *SSCA2*) show performance improvement when applying fine-grained resource allocation. Applications in the second group (*Vacation*, *Intruder*, and *Kmeans*) show limited or no performance improvement when fine-grained resource allocation is applied. This means that the load is well balanced between applications and auxiliary threads and that the original STM^2 design provides already high performance and processor utilization. For these applications, the adaptive solution aims at not worsening performance.

Statically decreasing AxT_p during embarrassingly parallel phases generally improves performance, up to 46% (*Bayes*) though the actual impact depends on the amount of time spend in these phases. Even in this case, static solutions may suffer from the

overhead of unnecessary changing the values of AT_p: for example, if the time between two transactions is short, the overhead of invoking a system call may outweigh the performance improvement obtained throughout embarrassingly parallel sections. This situation does not often arise with Eigenbench, where we have the complete control of the application's structure, but it appears in some of the STAMP benchmarks that present back-to-back transactions (e.g., *SSCA2* or *Vacation*). In some cases (e.g., *Intruder* and *Kmeans*) even simply reducing AxT_p between two consecutive transactions results in a considerable overall slowdown (10% and 5%, respectively). The same scenario arises when application threads are waiting at commit phase: if the auxiliary thread has already performed all the TM operations when the application thread reaches the commit phase, the system call overhead may induce performance degradation. Suspending idle auxiliary threads (*naïve-EP*) or spinning application threads (*naïve-Spin*) introduces an even larger overhead: in the worst cases (large number of small transactions) the performance slowdown can be up to 38% (*SSCA2*). In general, *naïve-EP* and *naïve-Spin* perform worst or equal than their hardware thread priority counterparts (*EP* and *Spin-only*, respectively). In order to avoid these situations, the adaptive solution snoozes for a short time before reducing the priority of waiting auxiliary threads during embarrassingly parallel phases or application threads waiting at commit phase. In particular, the adaptive solution reduces AT_p only if there are at least `SPIN_THRESHOLD` messages (20 in the current implementation) in the communication channel when an application thread reaches the commit phase. Similarly, the adaptive solution reduces AxT_p in embarrassingly parallel phases only after `EP_THRESHOLD` cycles.

Within transactions, static techniques do not usually provide performance improvement even for applications in the first group: Prioritizing auxiliary threads ($\Delta_p < 0$) always reduces application's performance while prioritizing application threads ($\Delta_p > 0$) provides speedups for *Bayes*, *Labyrinth* and *Yada*. Spinning at commit phase provides measurable performance improvement only for *Bayes*, which proves that auxiliary threads are not particularly overloaded, hence AxT_p should not generally be greater than AT_p (unless particular situations arise). For well-balanced applications, such as *Vacation*, static approaches provide performance degradation or have almost no effect (performance variation are within 1%). In these cases, the adaptive solution does not generally detect load imbalance and, therefore, does not react (i.e., no priority change) with the result that there is almost no performance variation.



(a) Labyrinth's transaction execution trace

(b) Labyrinth's execution trace: close-up of the final part of a transaction

Figure 6.10: Labyrinth's transactions alternate a large local computation phase (white in the figure) with a burst of transactional operations (colored bars) at the end. The adaptive solution is able to capture this structure and properly set AT_p and AxT_p in each sub-phase. The result is performance improvement of 19% over the standard STM^2.

Among the tested STAMP applications, *Labyrinth*, *SSCA2* and *Bayes* are the most interesting cases for this study. *Labyrinth* presents very large, back-to-back transactions with a large number of accesses to local memory locations followed by a burst of shared memory accesses (TM operations). This behavior is similar to the one presented in Section 6.3.2.2, where a burst of TM operations appears towards the end of a transaction, but more extreme compared to the modified version Eigenbench. In particular, both the local computation phases and the number of TM operations issued in burst are considerably larger. Figure 6.10a depicts the execution trace of one of *Labyrinth*'s transaction: the picture clearly shows that the local computation phase (white in the trace) is predominant, which explains why even statically reducing AxT_p provides some performance improvements (Figure 6.9). Figure 6.10b shows a close-up of the final part of the transaction, the burst of TM operations. Since the burst is at the end of the transaction, the application thread has to wait at commit phase for the auxiliary thread to complete all TM management operations, though the auxiliary thread is mainly idle during the transaction (which explains the small performance improvement for the spin-only case in Figure 6.9). As it was the case for the examples in Section 6.3.2.2, due to the non-uniform structure of the transaction, static solutions fail to capture the application's characteristics. The adaptive solution, instead, is able to lower the priority of the auxiliary thread in the first part of the transaction, reaching $AxT_p = 1$, and increase the auxiliary thread priority when the application thread issues the burst of TM operations. Overall the adaptive solution outperforms the standard STM^2 by 19%.

SSCA2 presents two separate execution phases: in the first phase, the application generates the graph that will be solved in the second phase. Both phases are parallel but, while the second phase uses transactions to protect shared memory locations, the first phase is embarrassingly parallel, as each thread works on its local portion of the graph. The original STM^2 assigns half the available hardware threads to run auxiliary threads even in the embarrassingly parallel phase: by statically reducing the priority of the auxiliary threads in the first phase, static solutions achieve 10.3% of performance improvement over the standard STM^2 design. In the second part of the application, *SSCA2* performs very short and balanced transactions with a low conflict rate and several concurrent writers. In this phase, the application is well balanced and applying static solutions decreases performance. The adaptive solution also lower AxT_p in the first phase but does not detect load imbalance within transactions, hence it does not react. The net result is a performance improvement of 9.8% over STM^2. For this application, suspending waiting application or auxiliary threads has a dramatic impact on performance caused by the large number of short transactions.

Bayes is the application that shows the largest performance improvement: Even static approaches achieve improvement in the order of 30-45%. *Bayes* implements an algorithm for learning the structure of Bayesian networks from observed data through a hill-climbing strategy. To this extent, the application combines local and global search. Similarly to *SSCA2*, the applications performs two parallel parts: the first is mainly embarrassingly parallel and devoting more hardware resources to application threads considerably increases performance (up to 45%). In the second part, instead, the applications uses a few large transactions with large read- and write-sets. However, similarly to *Labyrinth*, auxiliary threads are frequently idle, thus decreasing AxT_p provides benefits. For *Bayes* the adaptive solution precisely captures the application's structure and combine the positive effects observed for *Labyrinth* and *SSCA2*, providing a final performance improvement of 85% over the standard STM^2.

The examples shown in this section demonstrate that there are cases in which fine-grained hardware resource partitioning can be used to improve the performance of assisted execution systems, such as STM^2. For not well-balanced applications, like *Labyrinth* and *Bayes*, and for applications with large (sequential or parallel) independent computing phases, like *SSCA2* and *Bayes*, the adaptive solution successfully employs dynamic fine-grained hardware resource partitioning and provides considerable

performance improvements without any effort from the programmer, modification of the applications or library re-linking. Only *Intruder* among all the STAMP applications shows minimal performance degradation. Finally, considering that the STM^2 already outperforms several state-of-the-art STMs [82], the adaptive solution shows average speedup of 2.88x, 2.68x, 2.41x, and 6.21x over TinySTM [133], NOrec [39], TL2 [43], and TML [149], respectively.

6.6 Related work

Hardware thread prioritization [57, 107] has been introduced by IBM in the POWER5 processor family. Hardware thread prioritization allows users to dynamically bias the amount of hardware resources assigned to hardware threads in the same core. AIX [57] provides the users with an interface to modify hardware thread priorities. Linux kernels use hardware prioritization when 1) a thread is spinning on a lock, 2) a thread is waiting for another thread to complete a required operation (`smp_call_function()`), or 3) a thread is idle. Linux resets the priority of a thread after receiving an interrupt or an exception and does not keep a per-process priority status. Moreover, Linux does not consider the priority of the paired thread and, since the prioritization mechanism works with the priority difference, arbitrarily modifying the priority of one hardware thread may invalidate the decision taken on the other. Boneti et al. [17] characterized the use of hardware thread prioritization for POWER5 processors running micro-benchmarks and SPEC benchmarks. Other researchers [111] have also investigated the effect of hardware thread priorities on the execution time of co-scheduled application pairs on a trace-driven simulator of the POWER5 processor. Moreover, in a follow-up work, Boneti et al. used hardware prioritization to transparently balance high performance computing applications [18, 19], achieving up to 18% performance improvement.

Mann et al. [103] proposed a holistic approach that aims at reducing Operating System (OS) jitter by utilizing the additional threads or cores in a system. The authors tried to handle jitter through different approaches, one of the approaches is setting the hardware priority of the primary SMT thread to priority 6 and that of the secondary SMT thread to priority 1 in order to reduce jitter caused by SMT interference.

6.7 Conclusions

Assisted execution models can relieve application threads from the overhead of running runtime system functionalities and improve performance, even in those cases where the theoretical speedup computed with Amdahl's law does not justify the use of extra cores/hardware threads. However, assisted execution models, in general, present low processor utilization and the waste of resources.

In this work we propose to use adaptive fine-grained resource allocation to improve the efficiency and utilization of assisted execution models. We apply our solution to STM^2, a parallel software transactional memory system that offloads STM time-consuming operations to auxiliary threads. We propose an integrated hardware/software approach to implement fine-grained resource allocation for STM^2: our work spans the full hardware/software stack, from the hardware thread prioritization mechanism of IBM POWER7 processor to the programming language runtime system.

In order to understand the impact of fine-grained resource allocation on a complex system, such as STM^2, on real hardware, we followed a step-by-step approach in which we separately and statically apply different techniques to Eigenbench and STAMP applications. In the second phase, static techniques are integrated with heuristics that automatically detect computing power requirements and drive the hardware actuators to dynamically perform hardware resource allocation. The proposed adaptive solution improves performance and resource utilization for applications that prove to be challenging for the original STM^2. Results obtained on a state-of-the-art IBM POWER7 system with 32 hardware threads show that adaptive fine-grained resource allocation provides performance improvement up to 65% over the standard STM^2 design for Eigenbench, a simple and malleable TM benchmark, and up to 86% for more complex applications from the STAMP benchmark suite. Our experience with the IBM POWER7 hardware prioritization mechanism suggests that integrated hardware/software solution are interesting and can be employed to efficiently solve problems that may be difficult to solve completely at one level. However, a more fine-grained hardware prioritization mechanism that provides more intermediate values rather than extreme values, such as the current IBM POWER7 mechanism, would further help fine tuning integrated hardware/software solutions.

Finally, we remark that, although we applied fine-grained hardware resource allocation to STM^2, this approach can be used for other assisted execution systems, such as OS exception handlers [166] or dynamic check in Java Script [109].

Part IV

Correctness Semantics for TM applications

Although TM has reached maturity level and several STM and HTM implementations are available, there is still lack of debugging tools that automatically check the correctness of C/C++ TM programs, particularly race detection tools. The current definition of transactional data race requires all transactions to be totally ordered "as if" serialized by a global lock, which limits scalability of TM designs.

In Chapter 7, we revisit the current correctness model for TM applications, mainly those based on the happens-before relation, and analyze their strengths and weaknesses. We first propose to relax the current definition of transactional data race to allow a higher level of concurrency. Based on this relaxed definition, we propose the first practical race detection algorithm for C/C++ applications (TRADE) and implement the corresponding race detection tool.

Then, in Chapter 8, we propose a new definition of transactional data race that is more intuitive, is transparent to the underlying TM implementation, can be used for a broad set of C/C++ TM programs, enables a wide range of implementation techniques to be used, and allows the implementation of efficient dynamic race detection tools for TM applications. Based on this new definition, we propose *T-Rex*, an efficient and scalable race detection tool for C/C++ TM applications.

We analyze the precision and the performance of both tools and compare them with each other and with a race detection tool based on the current definition of transactional data race. Our experiments show that *T-Rex* and TRADE have discovered subtle transactional data races in a widely-used STAMP applications which have not been reported in the past.

Chapter 7

TRADE: Precise Dynamic Race Detection for Scalable Transactional Memory Systems

7.1 Introduction

The dominance of multi-core processors has made concurrent programming essential to achieve optimal performance. Unfortunately, despite the performance benefit, parallel programming introduces high software complexity and is prone to synchronization bugs, such as data races. There have been significant efforts to develop TM systems, hardware (HTM) [42, 63] and software (STM) [39, 43, 133], compilers with TM support [31, 34, 77] and basic debuggers for TM [69, 169]. However, there is not yet a consensus on a single definition for a data race for TM applications and there is no race detection tools that help programmers discover data race conditions in real C/C++ TM programs.

This lack of consensus on the definition of transactional data race and on the notion of what it means for a TM program to be correctly synchronized, motivated the development of several different correctness disciplines which constrain the behavior of correct TM applications, ranging from *Static Separation* (SS) in STM-Haskell [67] and *Dynamic Separation* (DS) in AME [3] to *Transactional Data Race Free* (TDRF) [38]. Under each of these disciplines, a correct implementation is required to provide the "fundamental property" [136] of memory models, which defines that a correctly synchronized program appears to run with a simple semantics that can be understood with

reference solely to the source language, rather than with reference to details of the implementation. For instance, with TM, this semantics would typically require that the programmer sees transactions with strict serializability, and without seeing the effects of speculative execution.

Most of the researchers have suggested that transactional semantics should be defined in terms of locking semantics [16], and that TM should be considered as a technique for implementing the semantics of a single global lock, but allowing greater concurrency than what would be implied by actually having a single lock [110]. This basic, pragmatic semantics is called *single global lock atomicity* (SGLA), where a program is required to behave "as if" transactions were protected by a single global lock. More precisely, a TM system is said to provide SGLA if, for every program execution, there exists some global total order on all transactions that is consistent with program order, and that when closed with program order produces a happens-before order that explains the program's reads. This semantics is equivalent to the one proposed by Dalessandro et al. [38] in TDRF model, Grossman et al. [58] and the one currently adopted by the Draft Specification of Transactional Language Constructs for C++ [6].

SGLA semantics also leads to a definition of what it means for a program using transactions to have a data race: the transactional program has a data race if and only if the equivalent lock-based program has a data race. Assuming that an underlying STM implements SGLA semantics, researchers have extended the definition of data race based on the happens-before relation for lock-based applications to transactional memory applications [38, 58, 110]. In this work, we call this relation *strict transactional happens-before*.

Although SGLA simplifies the design, implementation and testing of STM systems, many important scalable STMs do not implement SGLA semantics. The reason is that forcing a total order among all transactions may reduce the concurrency level, which is not desirable from the performance point of view. Performance (scalability in particular) is of paramount importance for modern multi-core systems, hence programmers have been reluctant to use STM designs that implement SGLA semantics. For example, TinySTM [133], NORec [39], and TL2 [43] are commonly used STMs, yet none of these STMs implements SGLA semantics. Hence, the applicability of race detection tools based on the strict happens-before relation is limited.

In order to remove the constraints introduced by SGLA semantics on the implementations and to be able to run TM applications on scalable STMs, we relaxed the definition of strict happen-before by requiring total ordering only among conflicting transactions (*relaxed happens-before relation*). We denote two transactions as *conflicting* if there exists a memory access to `x` from two different threads and one transaction writes to and the other either reads from or writes to `x`. If no such conflicting access exists, the transactions do not have to be ordered with respect to each other (*non-conflicting*). Based on the relaxed happens-before relation, we propose Transactional data RAce DEtection (TRADE), a novel and precise race detection algorithm for TM applications. The algorithm determines whether an execution of a TM application is race-free by tracking relaxed happens-before edges among conflicting transactions. Based on TRADE, we implement a dynamic race detection tool for C/C++ TM applications.

We also design a race detection algorithm for TM applications running on STMs that give SGLA semantics (s-TRADE) and implement the corresponding dynamic race detection tool. This tool requires the underlying STM to implement SGLA as suggested in [110] and cannot be used with an STM that does not enforce total ordering among all transactions, such as TinySTM.

We analyze the precision of each dynamic race detectors on a 8-core Intel Nehalem system with STAMP applications, a benchmark suite commonly used to test STM systems [22]. Although STAMP benchmarks are considered to be mature applications, our tools detect potentially harmful transactional data races that, to the best of our knowledge, have not been previously reported. To further analyze the soundness of our tools, we inject data race bugs into STAMP applications and verify that they are precisely detected. We also compare the race detectors in term of performance: Despite the fact that TRADE requires more work than s-TRADE to establish relaxed happens-before edges among conflicting transactions, TRADE runtime overhead with respect to s-TRADE is generally negligible. More importantly, TRADE can be used with scalable STM systems that provide higher performance compared to STMs that implement SGLA semantics.

In this chapter we make the following novel contributions:

- We propose two novel race detection algorithms for TM applications. To the best of our knowledge, these are the only practical algorithms proposed in the

literature. TRADE, in particular, can be used with many high-performance STMs.

- Based on our algorithms, we implement the corresponding dynamic race detection tools for real C/C++ TM applications.
- We analyze STAMP applications and discover potentially harmful transactional data races for *SSCA2* that have not been reported in the past.

This chapter is organized as follows: Section 7.2 provides information about the properties of TM systems, their implications and their relation with correctness models. Section 7.3 reviews preliminary concepts of strict and relaxed happens-before relation. Section 7.4 and 7.5 details our transactional race detection algorithms and implementations, respectively. Section 7.6 evaluates our race detection tools in term of precision and performance. Section 7.7 presents the related work. Section 7.8 concludes this chapter.

7.2 Background

Although the definition of data race is orthogonal to the synchronization mechanism, critical sections protected by locks and transactions are semantically different and present distinct characteristics and requirements, thus race detection tools used for lock-based applications cannot be directly extended to transactional memory. With TM, a transaction either executes completely and atomically or should appear as if it were never executed. This means that transactions' effects should be permanently visible, and thus can generate data races, when a transaction successfully commits. In fact, a transaction may abort, causing all modified memory locations to roll back to their original values, as if they had never been modified. With lock-based applications, instead, memory locations modified by a thread inside a critical section are immediately and permanently visible to other threads and can, therefore, immediately generate data races.

TM systems guarantee that threads have a consistent view of the memory among transactions, eventually aborting conflicting transactions. TM systems with support for *weak isolation* [106] guarantee transactional semantics only among transactions, i.e., accesses to shared memory locations within transactions appear as atomic operations

Thread 1	Thread 2
`atomic {`	`tmp1 = ready;`
`  data = 42;`	`tmp2 = data;`
`  ready = true;`	`if tmp1 {`
`}`	`  tmp2++;`
	`}`

Figure 7.1: Does `ready=true` imply that `Thread 2` sees `data=42`?

to other transactions. TM systems with support for *strong isolation* [15], instead, also guarantee transactional semantics between transactional and non-transactional code, hence normal non-transactional accesses are serialized by the TM with any concurrent transactions. Many HTM implementations naturally provide strong isolation, and there has been substantial progress in developing STMs that support strong isolation [2, 140, 143]. However, strong isolation requires extra instrumentation barriers that introduce large runtime overhead, especially on STM designs. Most of the state-of-the-art STMs only support weak isolation and rely on the programmer or race detection tools to guarantee that the program is correctly synchronized.

Moreover, even if the implementation of an STM system provides strong isolation, the system still needs to account for the interaction between transactional and non-transactional accesses. Consider the example in Figure 7.1: the intent of the programmer is to prepare some data and publish it once it is ready, thus when `ready` is `true`, the programmer expects `Thread 2` to see `data = 42`. However, although the correctness of the transactional code is guaranteed by strong isolation, a programmer cannot assume that, if `Thread 2` sees `ready = true` then it must also see `data = 42`. This line of reasoning is only correct if `Thread 2`'s implementation is guaranteed to read from `ready` before it reads from `data` (sequential consistency). This ordering is not enforced by many programming languages (e.g., Java) or by processors with weak memory models (e.g., POWER processors) and, since there is no explicit dependency between `data` and `ready`, a compiler can apply reordering optimization [38]. In other words, strong isolation does not imply sequential consistency. Therefore, programs running on TM systems with strong isolation may still incur data races.

7.3 Preliminaries

This section defines transactional data races based on the previous definition of strict happens-before relation and on our relaxed transactional happens-before definition. We represent an execution of a multi-threaded program as a sequence of actions, such as transactional/nontransactional read/write operations, begin/end transaction, fork/join and barriers and we assume the following:

- Accesses from an individual thread are ordered by *program order* ($\longrightarrow_{po}$).
- Transactions commit in a global temporal order, *commit order* ($\longrightarrow_{co}$). A transaction TX_i is ordered before a transaction TX_j, if TX_i commits before TX_j in the program execution.
- Synchronization primitives, such as `barrier()`, `fork()`, and `join()`, introduce a *sync-primitive order* ($\longrightarrow_{so}$). An access a performed before a synchronization primitive is ordered before an access b performed after that synchronization primitive.

In this work we consider applications that only use transactions to synchronize accesses to shared memory locations. While other synchronization mechanism (e.g., locks) could be used in conjunction with transactions, there is currently no agreed-upon correct semantics of programs that simultaneously use transactions and locks [64]. Moreover, we are not aware of any publicly available applications programmed with transactions and locks that we can use in our experiments. We leave this for future work.

7.3.1 Strict Transactional Happens-Before Relation

Although locks and transactions are semantically different, previous work focused on extending existing definitions of data race for lock-based applications to transactional memory applications. Researchers have investigated forms of *single global lock atomicity* (SGLA) and relaxed forms of this which map correct synchronization of TM programs into existing lock-based disciplines [38, 58, 110]. In STM systems that implements SGLA semantics, transactions are considered "as if" they were executed under a single global lock. Menon et al. explore the implementation and performance implications

of SGLA and the merits of various definitions for which pairs of transactions are ordered [110]. They show that a sufficient condition for SGLA is to allow concurrent execution of transactions but to linearize their execution at commit phase in a staggered, pipelined fashion. This staggered execution of transactions provides an explicit total ordering over all transactions:

Definition 1. All transactions in the execution are totally ordered by a *strict synchronized-with relation* ($\longrightarrow_{ssw}$) if and only if they are ordered by commit order. If transaction $A \longrightarrow_{ssw} B$ then access a in $A \longrightarrow_{ssw}$ access b in B.

With the strict synchronized-with relation, two transactions are allowed to run in parallel but they need to commit in linear order, even if their read- and write-sets do not overlap.

Definition 2. The irreflexive transitive closure of program order, sync-primitive order and the strict synchronized-with relation define a *strict transactional happens-before* ($\longrightarrow_{shb}$) partial ordering on all accesses in the execution.

Definition 3. A *strict transactional data race* exists between two accesses in a given execution if and only if they access the same location, at least one is a write, they are executed by different threads, and are not ordered by $\longrightarrow_{shb}$.

This definition is equivalent to the ones proposed in previous work [38, 58, 110] and the Draft Specification of Transactional Language Constructs for C++ [6].

7.3.2 Relaxed Transactional Happens-Before Relation

SGLA limits the nature of TM by not allowing non-conflicting transactions to commit concurrently. The result is that STMs that implement SGLA provide much lower performance than STMs that provide higher concurrency.

Figure 7.2 shows speedup of several popular STM designs over sequential version. The experiments are conducted with STAMP applications running with eight threads on a 8-core Intel Nehalem system. In the graph we use STM implementations from the Rochester Software Transactional Memory package (RSTM) [147]: LLT is a lazy conflict detection, write-buffered design similar to TL2 [43]; ET is a eager conflict detection, write-buffered design inspired by TinySTM [133]; NORec [39] is a lazy conflict detection, write-buffered STM based on a single sequential lock; finally Pipeline is a

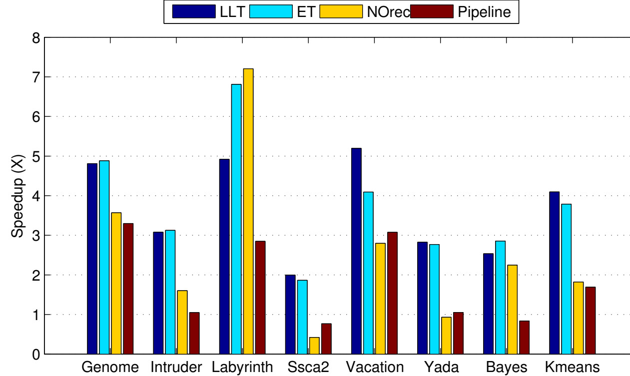


Figure 7.2: Speedup of STAMP applications with various STMs.

lazy conflict detection, write-buffered STM that implements SGLA semantics as described by Menon et al. [110]. All the STMs tested, except Pipeline, allow read-only transactions to commit in parallel; LLT and ET also allow read-write transactions that have no conflicts to commit in parallel. The graph clearly shows that the performance of Pipeline is considerably lower than the other STMs: in some cases, such as *Bayes* and *Intruder*, Pipeline performance with eight threads barely matches the single threaded execution. On the other hand, LLT and ET achieve good speedups for most of the cases. Because of these performance reasons, programmers have been averse to using STMs with SGLA semantics. Many important state-of-the-art STM designs do not implement SGLA semantics: TinySTM [133] (recently used in GCC-TM [138]), NORec [39] (which allows read-only transactions to commit in parallel) and TL2 [43]. This, in turn, limits the applicability of race detection tools based on the strict transactional happens-before relation.

Moreover, the strict transactional data race definition may produce results that do not fit with programmers' intuitive expectations for transactions' execution. Consider the example in Figure 7.3: This program is racy, as `Thread1` and `Thread2` may access `x` concurrently. Indeed, if we consider a program execution in which `Thread2`'s transaction (TX_2) commits before `Thread1`'s transaction (TX_1), then accesses to `x` in TX_1 and in `r1 = x` are not ordered by $\longrightarrow_{shb}$, hence there is a strict transactional data race, which matches the programmer's reasoning of the correctness of this program. However, if we consider a program execution in which TX_1 commits before TX_2, then

Thread 1	Thread 2
atomic {	atomic {
x = 1;	y = 42;
}	}
	r1 = x

Figure 7.3: This program has a transactional data race, as `Thread1` and `Thread2` may access x concurrently.

$TX_1 \longrightarrow_{shb} TX_2$. By program order $TX_2 \longrightarrow_{shb}$ `r1 = x`, hence $TX_1 \longrightarrow_{shb}$ `r1 = x`. It follows that, in this execution, there is no strict transactional data race, which contradicts the programmer's expectation. Note that, since TX_1 and TX_2 do not conflict, a programmer expects the two transactions to run and commit in parallel. However, this program execution is forbidden under STMs that implement SGLA.

In order to remove the excessive constraints and limitation imposed by the strict transactional data race definition, we propose an alternative definition of transactional data race that is more intuitive and can be used with a broader set of high-performance STMs. To this extent, we provide the following definitions to include less than the full atomic order:

Definition 4. All conflicting transactions in the execution are ordered by a *relaxed synchronized-with relation* ($\longrightarrow_{rsw}$) if and only if they are ordered by commit order. If transaction $A \longrightarrow_{rsw} B$ then access a in $A \longrightarrow_{rsw}$ access b in B.

Definition 5. The irreflexive transitive closure of program order, sync-primitive order and the relaxed synchronized-with relation define a *relaxed transactional happens-before* ($\longrightarrow_{rhb}$) partial ordering on all accesses in the execution.

As opposed to the strict happens-before relation, two non-conflicting transactions are not ordered by the relaxed happens-before relation.

Definition 6. A *relaxed transactional data race* exists between two accesses in a given execution if and only if they access the same location, at least one is a write, they are executed by different threads, and are not ordered by $\longrightarrow_{rhb}$.

This definition of relaxed transactional data race can be used with STMs that do not implement SGLA semantics because it does not assume transactional total ordering among all transactions. In the particular example in Figure 7.3, `r1 = x` in `Thread2`

and the access to x in Thread1 are not ordered by $\longrightarrow_{rhb}$, hence there is a relaxed transactional data race when the two threads access x regardless of the order in which transactions commit. Note that, the programmer's expected execution in which TX_1 and TX_2 run and commit in parallel is also allowed by the underlying STM: This program execution also presents in a relaxed transactional data race.

Comparison: Two transactions ordered by relaxed happens-before relation are also ordered by strict happens-before relation, i.e., $TX_1 \longrightarrow_{rhb} TX_2 \Rightarrow TX_1 \longrightarrow_{shb} TX_2$, but not vice-versa. Let us define E as the set of strict happens-before edges in a given execution and E' as the set of relaxed happens-before edges in the same execution. Since the set of conflicting transactions is a subset of all transactions, it follows that $E' \subseteq E$. The other direction is not true: there are transactions ordered by strict happens-before relation that are not ordered by relaxed happens-before relation (e.g., read-only transactions), then $TX_1 \longrightarrow_{shb} TX_2 \not\Rightarrow TX_1 \longrightarrow_{rhb} TX_2$ because $E \nsubseteq E'$. Summarizing, all transactional data races detected by an algorithm based on $\longrightarrow_{shb}$ are also detected by an algorithm based on $\longrightarrow_{rhb}$ but not vice-versa.

Privatization-safety: Happens-before relations (for example, both the strict and the relaxed happens-before relations defined above) handle many styles of programming, including privatization and publication, two techniques used by programmers to directly access shared objects that are temporarily private to a thread. However, as explained in Section 8.2, the underlying STM must implement a form of *privatization-safety* (which usually requires inserting memory barriers and global synchronization among all running threads). Because of performance issues, only a few STMs, among the commonly-used state-of-the-art STMs (see Section 2.2.4) implement privatization-safety, i.e., IntelSTM, NORec. Finally, many STMs, e.g., TinySTM, TL2, do not suport privatization-safety. In order to use a race detection algorithm based on a happens-before relation, the STM should either be extended to support privatization-safety (as explained for TL2 in [110]) or the programmer must ensure correctness through (implicit or explicit) synchronization barriers. Privatization-safe versions of STMs that are not originally designed to be privatization-safe (e.g., TL2) are not usually available. In this work, thus, we rely on the programmer to guarantee privatization correctness of applications that use privatization/publication idioms when running on a STM that does

not implement privatization-safety, as the ones used in Section 7.6. This, of course, poses a constraint on the programmer. We will remove this constraint in the next chapter, where we introduce a new correctness model that is not based on any forms of happens-before relation and does not assume that an STM implements privatization-safety.

7.4 Transactional Race Detection Algorithms

In this section we describe our race detection algorithms for TM applications. The first algorithm (s-TRADE) detects transactional data races based on the strict transactional happens-before relation (Section 7.4.1). This algorithm is designed for STM systems that implement SGLA, as it assumes a form of total ordering among all transactions in the system. The second algorithm (TRADE), instead, is based on the relaxed transactional happens-before relation and does not assume SGLA, hence it covers a broader set of STM implementations (Section 7.4.2). The correctness proofs of the algorithms (soundness and completeness) are presented in Appendix A.

Both algorithms are based on *vector clocks*: a vector clock $VC : Tid \rightarrow Clk$ records a clock for each thread $t \in Tid$, where Tid is the set of all threads in the system. For each thread $t \in Tid$, we define the following operations:

$$
\begin{array}{rcr}
V_1 \sqsubseteq V_2 & \texttt{iff} & \forall t \in Tid, V_1(t) \leq V_2(t) \\
V_1 \sqcup V_2 & = & \lambda t.max(V_1(t), V_2(t)) \\
\bot_V & = & \lambda t.0 \\
inc_t(V) & = & \lambda u.\ \texttt{if}\ u = t\ \texttt{then}\ V(u) + 1\ \texttt{else}\ V(u)
\end{array}
$$

Vector clocks are partially ordered ($\sqsubseteq$) in a point-wise manner, with an associated join operation ($\sqcup$) and minimal element ($\bot$).

The semantics of transactional memory requires transactions to be executed entirely and atomically and their effects to be permanent only after successful commits. On abort, all transactional operations appear as if they were never executed and their effects (writes to memory) should not be visible to other threads. Atomicity is achieved by two

possible designs that differ in the way the STM updates modified memory locations: *in-place* and *write-buffered*. For simplicity, we present our algorithms for write-buffered STMs (e.g., TL2), where transactional writes are buffered into a local data structure and the memory locations modified by a transaction are only updated upon successful validation of the transaction at commit phase. However, we highlight here the main difference between in-place and write-buffered STMs from a race-detection perspective.

In-place STMs optimistically update shared memory locations when a transactional write is issued; if the transaction aborts, the modified memory locations are rolled back to their original values. The main difference between in-place and write-buffered STMs is whether aborted transactions are considered part of the execution (*visible abort semantics*) or not (*invisible abort semantics*) [142]. In-place STMs usually adopt visible abort semantics, which means that transactional writes performed by an aborted transaction can still originate data races. In this case our algorithms must verify the occurrence of a data race at the moment a transactional operation is issued, even if the transaction eventually aborts. With write-buffered STMs, instead, aborted transactions are generally not part of the execution, thus transactional writes performed by aborted transactions do not originate data races. In this case, our algorithms perform race detection upon successful validation of a transaction at commit phase.

7.4.1 s-TRADE Race Detection Algorithm

For STM systems that implements SGLA semantics, total ordering is defined by $\longrightarrow_{ssw}$. For these STMs, we propose the following algorithm based on vector clocks to detect strict transactional data races. We define:

$$\begin{aligned} C &: Tid \rightarrow VC \\ W &: Var \rightarrow VC \\ R &: Var \rightarrow VC \\ G &: Var \rightarrow VC \end{aligned}$$

where C is the vector clock of each thread $t \in Tid$, W and R are the write and read vector clocks of a variable $v \in Var$ and G is the global vector clock used to establish strict transactional happens-before edges.

For each thread $t \in Tid$, the algorithm follows the next rules:

[**Thread Creation**]

$$\forall t \in Tid,\ i = 1..N \quad C_t(i) := 0$$
$$C_t(t) := 1$$

[***Before*** **Transaction Commit**]

$$C_t := C_t \sqcup G$$

[***After*** **Transaction Commit**]

$$G := C_t$$
$$inc_t(C_t)$$

[***TX / NonTX*** **Read Shared**]

$$R_x(t) := C_t(t)$$
$$W_x \sqsubseteq C_t \Rightarrow RaceFree$$

[***TX / NonTX*** **Write Shared**]

$$W_x(t) := C_t(t)$$
$$W_x \sqsubseteq C_t \quad and \quad R_x \sqsubseteq C_t \Rightarrow RaceFree$$

At thread creation, all entries in the vector clock of thread t (C_t) are initialized to 0, except the thread's clock ($C_t(t) = 1$). Strict transactional happens-before relations are established by G: at commit phase, before performing the STM writes to memory, the thread's vector clock is joined to the global clock G. After the transaction has been validated, the thread's vector clock is copied to G and then $C_t(t)$ is increased to record that now thread t has moved to the next clock.

For this algorithm, the rules for transactional and nontranscational read/write operations are the same. However, for write-buffered STMs that implement invisible abort semantics, the actual race detection for transactional operations is performed during the validation of the transaction at commit phase.[1]

[1]As explained before, race detection for in-place STMs that implement visible abort semantics is performed at the moment the transactional operation is issued, similarly to nontransactional operations.

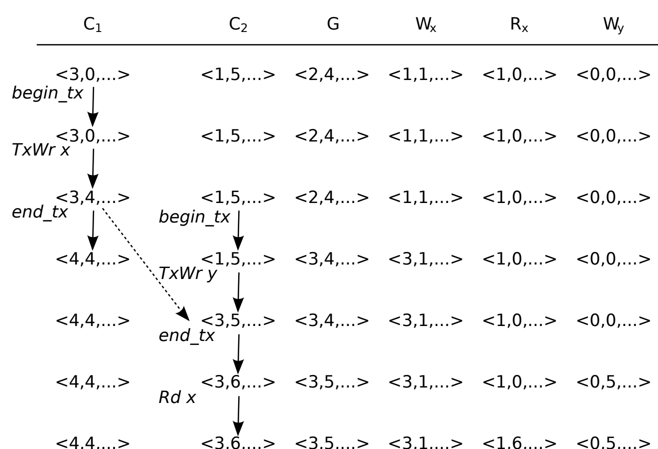


Figure 7.4: Example trace for the program in Figure 7.3 running on an STM that implements SGLA. No strict transactional data races detected.

Read operations For both transactional and nontransactional read operations, the variable x's read clock of the reading thread $R_x(t)$ is updated with the current thread's clock $C_t(t)$. Next, the algorithm searches for the occurrence of a possible *write-read* race: if all modifications to x are prior to the current thread's vector clock ($W_x \sqsubseteq C_t$), then there is no race detected.

Write operations Transactional and nontransactional writes to shared memory locations behave similarly to their respective read operations except that the algorithm also searches for *write-write* and *write-read* races: if all reads from x ($R_x \sqsubseteq C_t$) and all modifications to x ($W_x \sqsubseteq C_t$) are prior to the thread's vector clock, then there is no strict transactional race.

Figure 7.4 shows the evolution of s-TRADE race detection algorithm for the example shown in Figure 7.3, assuming that `Thread1`'s transaction commits before `Thread2`'s transaction. The figure shows that there is a strict happens-before edge between `Thread1`'s transaction and `Thread2`'s transaction. When `Thread2` accesses `x` nontransactionally, $W_x =< 3, 1, ... >$ and $C_2 =< 3, 6, ... >$, thus $W_x \sqsubseteq C_2$ and the execution is race-free.

7.4.2 TRADE Race Detection Algorithm

Unlike the previous algorithm, where producing strict happens-before edges only requires a single global vector clock, in this algorithm there are relaxed happens-before

edges between any two conflicting transactions, one edge for each variable. We say that a transaction TX_i is ordered with a transaction TX_j on a shared variable $x \in Var$ if they both access x and at least one access is a write. It follows that TX_i and TX_j can be ordered on x but not on y, if they do not conflict on y.

The relaxed transactional happens-before relation is established as follows: For each transactional read operation, a transaction TX has incoming edges from previous transactions that wrote the same memory location. Transactional write operations induce incoming edges between a transaction TX and previous transactions that read from or wrote to the same memory location. Conversely, there are outgoing edges from a transaction TX to subsequent transactions that read or write the memory locations accessed by TX. For each transactional read operation, there is an outgoing edge to subsequent transactions that write the same memory location. Similarly, each transaction write operation generates an outgoing edge to transactions that read or write the same variable. It follows that non-conflicting transactions (e.g., read-only transactions) are not ordered by the relaxed transactional happens-before relation.

For this algorithm, we define:

$$
\begin{aligned}
C &: Tid \rightarrow VC \\
W &: Var \rightarrow VC \\
R &: Var \rightarrow VC \\
TW &: Var \rightarrow VC \\
TR &: Var \rightarrow VC
\end{aligned}
$$

where C contains the vector clocks of each thread $t \in Tid$; W and R are the write and read clocks of a variable $v \in Var$, respectively; TR and TW contain the read and write transactional dependency clocks of each variable $v \in Var$. TR and TW are used to establish relaxed transactional happens-before relations. $\forall t \in Tid$, the algorithm follows the next rules:

[**Thread Creation**]

$$
\begin{aligned}
i = 1..N \quad C_t(i) &:= 0 \\
C_t(t) &:= 1
\end{aligned}
$$

[**Transaction Commit**]

$$inc_t(C_t)$$

[***NonTX* Read Shared**]

$$R_x(t) := C_t(t)$$
$$W_x \sqsubseteq C_t \Rightarrow RaceFree$$

[***TX* Read Shared**]

$$C_t := C_t \sqcup TW_x$$
$$R_x(t) := C_t(t)$$
$$TR_x(t) := C_t(t)$$
$$W_x \sqsubseteq C_t \Rightarrow RaceFree$$

[***NonTX* Write Shared**]

$$W_x(t) := C_t(t)$$
$$W_x \sqsubseteq C_t \quad and \quad R_x \sqsubseteq C_t \Rightarrow RaceFree$$

[***TX* Write Shared**]

$$C_t := C_t \sqcup TW_x \sqcup TR_x$$
$$W_x(t) := C_t(t)$$
$$TW_x(t) := C_t(t)$$
$$W_x \sqsubseteq C_t \quad and \quad R_x \sqsubseteq C_t \Rightarrow RaceFree$$

At the beginning of the application, the vector clock of each thread $t \in Tid$ is initialized to 0. When a thread t is created, its clock $C_t(t)$ is set to the value 1. C_t keeps track of thread t's clock and the clocks of any other thread $t' \neq t$ last observed by t. $C_t(t)$ is updated at the end of every transaction. The read and write vector clocks of a variable x, R and W, are initialized to 0 the first time x is accessed for reading or writing, respectively. TR and TW are also initialized to 0 the first time a variable is accessed transactionally.

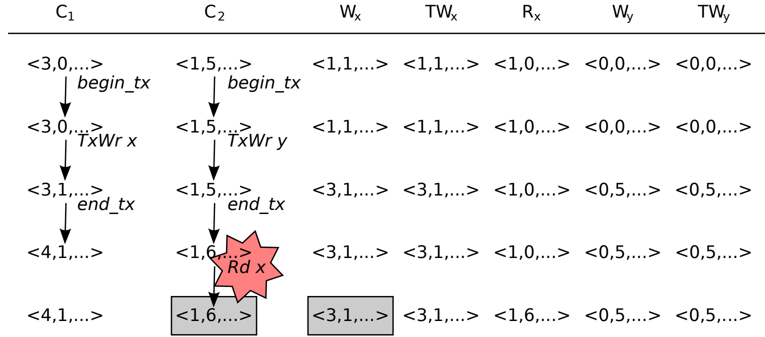


Figure 7.5: Example trace for the program in Figure 7.3 running on an STM that does not implement SGLA semantics. HB detects a transactional data race in a given execution.

Read operations Nontransactional read operations update the t-th entry of a variable x's read clock with the current thread clock ($R_x(t) := C_t(t)$) and then check whether a *write-read* race has occurred: If all writes to x are precedent to the last observed vector clock of thread t ($W_x \sqsubseteq C_t$), then there is no race. Transactional read operations first update the thread vector clock with the write transactional dependency clock ($C_t := C_t \sqcup TW_x$). This operation builds incoming edges between the current read operation and previous transactional write operations to x. Next, TR_x is updated to build outgoing edges between the current transaction and any subsequent transaction that writes x. As for nontransactional read operation, transactional read operations update the read vector clock of the variable x (R_x) and perform race detection.

Write operations Nontransactional write operations update the variable's write clock W_x with the value of the thread's clock ($W_x(t) := C_t(t)$) and then check for a possible occurrence of a data race. For write operations, both *read-write* and *write-write* transactional race conditions must be checked. On transactional writes, the thread vector clock is updated with both read and write transactional dependency vector clocks ($C_t := C_t \sqcup TW_x \sqcup TR_x$): this operation builds incoming edges from previous transactional that read or wrote x and the current transaction. Next, TW_x is updated with the current thread clock: this operation builds outgoing edges between the current transaction and any subsequent transaction that reads or writes x. Finally, transactional write operations update the write vector clock W_x and check if the current transactional write has generated any *write-write* or *write-read* transactional data race.

Figure 7.5 shows the evolution of TRADE algorithm for the example in Figure 7.3. In this example, `Thread1`'s and `Thread2`'s transactions do not conflict and are thus allowed to run and commit concurrently. Since the two transactions do not conflict, there are no relaxed happens-before edges between them. It follows that when `Thread2` accesses `x` nontransactionally, there is a possibility of a concurrent accesses with `Thread1`'s transactional access to `x`, thus the algorithm detects a transactional data race. More specifically, at the time `Thread2` accesses `x`, $W_X =< 3, 1, ... >$ while $C_2 =< 1, 6, ... >$, thus the algorithm detects a data race.

7.4.3 Extensions

Besides transactions and read/write accesses, our algorithms also track common operations that induce a partial/global ordering, such as thread creation/destruction and global barriers. Our algorithms need only track barrier releases, which indicate that all threads have reached the barrier.[1] The vector clock of a thread is updated with the current thread clock of each thread (maximum across all threads' vector clocks) and then all threads move to the next clock by adding one to each entry of their vector clock:

[**Barrier Release**]

$$\forall t \in Tid,\ C_t = inc_t(\bigsqcup_{u \in Tid} C_u)$$

Fork and join operations also introduce a partial ordering between parents and children. Thread fork/join operations follow the next rules, where we assume that thread t is the father of thread u:

[**Fork**]

$$C_u := C_u \sqcup C_t$$

$$C_t := inc_t(C_t)$$

[1] In case only a subset of threads participate to the barrier, we also need to track barrier entries and record which thread is involved in the barrier and should update its clock.

[**Join**]

$$C_t := C_t \sqcup C_u$$
$$C_u := inc_u(C_u)$$

7.5 Design and Implementation

This section presents the design and implementation of TRADE algorithm for transactional C/C++ applications.[1] Our prototype implementation of this algorithm checks if a TM program has relaxed transactional data races in a particular execution. If the tool detects relaxed transactional data races, it reports the address and the instruction of each transactional data race detected.

7.5.1 Binary instrumentation Framework

Dynamic race detection tools imply tracking accesses to shared memory locations. For unmanaged languages such as C/C++, current compilers for TM applications only provide automatic instrumentation of transactional accesses [6, 34]. None of them, to the best of our knowledge, instruments nontransactional accesses or provides hooks for dynamic checking tools. Manual instrumentation, on the other hand, is prohibitive for large, real applications such as the ones tested in this work. We, thus, implemented TRADE on top of Pin [100], a dynamic instrumentation tool that allows programmers to instrument transactional and nontransactional read and write accesses, as well as functions' entry and exit points. Pin enables users to dynamically modify binary applications on the fly, with no static annotation inserted by the programmer and no need of re-compiling/re-linking applications.

7.5.2 TRADE Instrumentation State and Code

In order to distinguish transaction from nontransactional code, we instrument begin/end transaction. `PIN_TM_BEGIN()` is invoked before the execution of the transaction:

[1] For clarity we present here the implementation for write-buffered systems with invisible abort semantics. The implementation for in-place systems with visible abort semantics is straightforward from the modifications discussed in Section 7.4. We also omit s-TRADE implementation description for the same reasons.

```
class ThreadState {
  int C[];
  int tid;
  List TxRead_list;
  List TxWrite_list;
}
class VarState {
  int R[];
  int W[];
  int TxR[];
  int TxW[];
}
void TXread(VarState x, ThreadState t){
  t->TxRead_list.insert(x);
}
void TXwrite(VarState x, ThreadState t){
  t->TxWrite_list.insert(x);
}
void commit(ThreadState t){
  for each element in t->TxRead_list
    eff_TXread(t->TxRead_list.remove(), t)

  for each element in t->TxWrite_list
    eff_TXwrite(t->TxWrite_list.remove(), t)
}
void eff_TXread(VarState x, ThreadState t){
  vc_cup(t.C, x.TxW);
  x.R[t.tid] = t.C[t.tid];
  x.TxR[t.tid] = t.C[t.tid];
  //write-read race?
  if (x.W[u] > t.C[u] for any u) Race!;
}
void eff_TXwrite(VarState x, ThreadState t){
  vc_cup(t.C, x.TxW);
  vc_cup(t.C, x.TxR);
  x.W[t.tid] = t.C[t.tid];
  x.TxW[t.tid] = t.C[t.tid];
  //read-write race?
  if (x.R[u] > t.C[u] for any u) Race!;
  //write-write race?
  if (x.W[u] > t.C[u] for any u) Race!;
}
```

Figure 7.6: Implementation of the TRADE algorithm.

the function sets a flag indicating that the thread has now entered a transaction. At commit stage, the STM library checks whether there are unresolved conflicts and, if not, the transaction commits and `PIN_TM_END()` is invoked. This function checks for the occurrence of race conditions for each memory locations in the read-/write-set and clears the flag. Note that, on abort, the transaction restarts from the beginning, thus, `PIN_TM_END()` is not invoked.

TRADE associates a `ThreadState` structure to each thread (see Figure 7.6). This structure contains a unique thread identifier `tid` and a vector clock `C`. Since a thread vector clock `C` is private to each thread and there are no concurrent accesses to the particular thread vector clock, there is no need to protect thread vector clocks with any lock. Each memory access has an association with `VarState` containing read and write vector clocks, `R` and `W`, respectively. Besides `R` and `W`, TRADE requires `TxR` and `TxW` to be able to establish relaxed transactional happens-before edges. Unlike thread vector clocks, transactional and nontransactional read and write vector clocks are accessed by many threads concurrently and must be protected by locks: we use fine-grained read/write locking (each read/write vector clock is protected by a specific lock) so that multiple threads can access disjoint vector clocks in parallel.

Figure 7.6 shows the most important TRADE event handlers, such as `TRADE_TXread()` and `TRADE_TXwrite()` are used to track transactional read/write accesses, respectively. TRADE is transparent to the underlying STM design and does not rely on the particular STM implementation or data structures (e.g., the read- and write-set). When a transactional operation is issued, we use shadow temporal read/write data structures (linked list without duplicates) to record memory locations accessed within a transaction. The semantics of transactional memory implies that modified memory locations are permanently visible to other threads only once the transaction has committed, thus we perform lazy race detection at commit phase. In more details, for each transactional read/write operations, the algorithm establishes relaxed transactional happens-before edges between the current and the preceding transactions: `TRADE_lazy_TXread()` and `TRADE_lazy_TXwrite()` update the thread vector clock with the maximum clock values between the thread vector clock (`C`) and `TxR` and/or `TxW`. Moreover, `TxR` and `TxW` are updated with the clock of the thread to establish relaxed transactional happens-before edges with following conflicting transactions. If the tool detects a write-read

Appl.	# Detected Races TRADE		# Detected Races s-TRADE	
	w/o Bug inj.	w/ Bug inj.	w/o Bug inj.	w/ Bug inj.
Intruder	0	1,705*	0	1,221*
Ssca2	99	102	99	102
Kmeans	0	253,044	0	238,457
Vacation	0	6,733	0	68
Genome	0	73	0	10
Yada	0	6	0	6
Bayes	0	3	0	0
Labyrinth	0	5	0	5

Table 7.1: Number of transactional data races detected by TRADE and s-TRADE without and with bug-injection. **Intruder* crashed because of the injected bug.

race (`W>C`) for transactional reads or read-write/write-write races (`R>C`/`W>C`) for transactional writes, it raises an alarm. Note that it is not necessary to check race-freedom conditions for repeated accesses to a variable within a transaction: because of the TM atomicity property, all accesses to the same variable in a transaction have the same thread vector clock ($C_t(t)$) value. For example, only the last transactional write to a variable `x` is checked regardless of how many transactional writes to `x` have been performed within the transaction.

The pseudocode for nontransactional read/write operations is exactly the same as their equivalent lazy transactional reads/writes, except for the operations that require `TxR` and `TxW` modification.

7.6 Evaluation

We validate the effectiveness of s-TRADE and TRADE through precision and performance analysis. We run applications from the STAMP benchmark suite, which is widely used to test TM systems, and we choose Pipeline and LLT from the RSTM library to have STMs with and without SGLA semantics, respectively. Note that s-TRADE can only be used with Pipeline because it requires SGLA; TRADE, instead, can be used with both Pipeline and LLT. The STAMP applications and the tested STMs are compiled with gcc 4.4.5 with optimization level O3 for 64-bit architectures, and run on an Intel Nehalem system (8 cores, 16GB of RAM). In all the experiments

```
thread_barrier_wait();
for (i = i_start; i < i_stop; i++) {
    for (j = inVerId[i];
         j < (inVerId[i] + Degree[i]);
         j++) {
           if ((j - inVerId[i]) < MAX_SIZE) {
            inVerList[j] =
                  List[i*MAX_SIZE+j-inVerId[i]];
        } else {
            inVerList[j] =
              auxArr[i][(j-inVerId[i]) % MAX_SIZE];
        }
    }
}
thread_barrier_wait();
```

Figure 7.7: SSCA2 code snapshot.

we run eight threads. To ensure a fair comparison, both tools are implemented on top of Pin and as similarly as possible.

Precision Analysis: Table 7.1 reports the number of transactional data races detected by TRADE running with LLT and s-TRADE running with Pipeline with and without bug injection.

The second and forth columns in the table show the number of transactional data races for the unmodified version of STAMP applications. Even though STAMP benchmarks are mature applications, both our tools detect transactional data races for *SSCA2*. To the best of our knowledge, these real transactional data races have not been previously reported.

SSCA2 consists of four kernels that work on a large, directed, weighted multi-graph and includes a scalable data generator that produces edge tuples containing a start and end vertex and the weight of each directed edge. The transactional implementation focuses on kernel 1 (which builds the graph) and presents short transactions with small read-/write-sets. The large number of nodes in the graph leads to infrequent conflicts. Moreover, in several parts, each thread works on a disjoint partition of the graph, without synchronizing the accesses to the nodes. Both s-TRADE and TRADE detect transactional data races in the process of creating the inner vertex list (`inVerList`). This code is enclosed between two global barriers and no transactions are used to protect

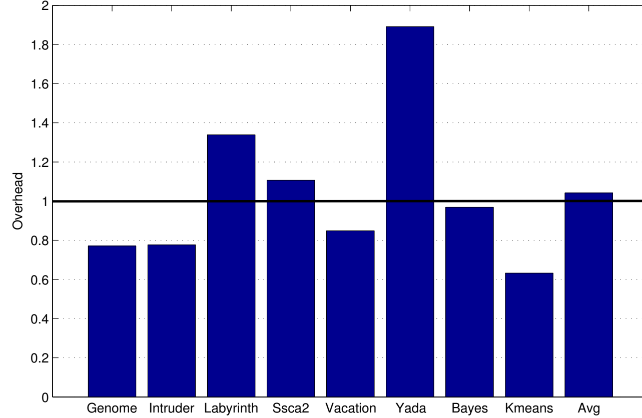


Figure 7.8: TRADE runtime overhead over s-TRADE.

accesses to the graph's nodes, hence the accesses to the nodes are ordered neither by $\longrightarrow_{shb}$ nor by $\longrightarrow_{rhb}$. Figure 7.7 shows the snapshot of the code where the transactional data races occur. During the creation of the inner vertex list, each thread accesses the next `Degree[i] > 0` vertices of a vertex with Id `inVertId[i]`, where `Degree[i]` is the degree of vertex `i`. This means that thread t_k accesses the first nodes in thread t_{k+1} partition when analyzing the last nodes (`i = i_stop-1`) in its partition without synchronization. Although we have not experienced any incorrect result or crash in our tests, as the number of threads increases, and therefore the number of nodes in the threads' partitions decreases, there is a higher probability that these transactional data races will cause a serious error.

To verify that TRADE and s-TRADE are sound race detection algorithms, we inject bugs into STAMP applications in the form of removing transactions, which transforms transactional sections into nontransactional ones. We then manually check that the tools detect all and only the injected data races. The third and the fifth columns in Table 7.1 show the number of transactional data races detected by TRADE and s-TRADE when injected bugs. As described in Section 7.3, all happens-before edges produced by TRADE are also produced by s-TRADE but not vice-versa. In more details, read-only and non-conflicting transactions are ordered by $\longrightarrow_{shb}$ but not ordered by $\longrightarrow_{rhb}$. We, thus, expect the number of transactional data races detected by TRADE to be equal or greater than the number of transactional data races detected by

s-TRADE. Note that both tools are precise: simply using LLT allows non-conflicting transactions to commit in parallel while Pipeline does not.

Table 7.1 confirms our intuition: the number of transactional data races detected by TRADE is always greater or equal than the number of data races reported by s-TRADE. In particular, TRADE detects more data races than s-TRADE for *Kmeans*, *Vacation*, *Genome*, and *Bayes*. These applications perform read-only and/or non-conflicting transactions that can commit in parallel with LLT but not with Pipeline. For *SSCA2*, *Yada* and *Labyrinth* the removed transaction produces the same effect on both set of happens-before edges, hence the both tools detect the same transactional data races. For example, for *SSAC2* the removed transaction is enclosed between a pair of barriers while for *Yada* the memory locations in the removed transaction are accessed only in that portion of the code.

Performance Analysis: To make a fair comparison between s-TRADE and TRADE and evaluate the performance impact of tracking relaxed happens-before edges, we run both race detection tools on STAMP applications using Pipeline as the underlying STM. Figure 7.8 shows the runtime slowdown of TRADE with respect to s-TRADE. Intuitively, TRADE is expected to introduce larger overhead than s-TRADE because the tool needs to track potential relaxed happens-before edges between transactional read and write operations (nontransactional operations behave similarly). However, as we can see from Figure 7.8, TRADE generally introduces lower overhead than s-TRADE because there are other factors that also account for the total runtime overhead when running STAMP applications.

STMs that implements SGLA do not allow multiple threads to commit in parallel; s-TRADE poses the additional constraint that race detection must also be carried on as part of the commit operation. This is necessary to ensure that no other thread updates any read/write vector clocks while a thread is still performing race detection. On the other hand, the serialization time at commit phase increases and eventually worsens the performance. Serialization time depends on two parameters: i) the total number of transactions (second column in Table 7.2) and ii) the size of each transaction, computed as the average number of transactional accesses per transaction (eighth column in Table 7.2). Applications with a large number of transactions (such as *Kmeans* and *Intruder*) show, in general, better performance with TRADE than with s-TRADE. Although *Intruder* and *SSCA2* have comparable number of transactions, s-TRADE

Apps.	Transactions		#Transactional Accesses				Per Tx Accs.	#Nontransactional Accesses		TX/ Non Tx
	Total	Read Only	Total		Unique					
			Read	Write	Read	Write		Read	Write	
Intruder	6,045K	31.10%	55,752K	3,164K	9.85%	40.09%	9.74K	25,699K	10,338K	1.63
Ssca2	5,558K	0.00%	2,780K	5,560K	9.38%	54.69%	1.50K	157,486K	34,243K	0.04
Kmeans	10,207K	0.00%	13,113K	6,588K	0.01%	0.01%	1.93K	513,365K	1,507K	0.03
Vacation	2,097K	0.15%	288,957K	7,099K	4.68%	62.14%	141.18K	37,639K	14,645K	5.66
Genome	2,489K	56.55%	58,288K	1,638K	2.21%	80.24%	24.07K	10,392K	6,259K	3.59
Yada	30K	35.71%	1,647K	240K	18.71%	51.19%	62.92K	68K	57K	15.09
Bayes	2K	26.92%	33K	3K	2.57%	14.25%	18.00K	218K	19K	0.15
Labyrinth	1K	0.76%	92K	91K	98.05%	99.71%	183.00K	4K	3K	26.14

Table 7.2: STAMP applications' characteristics.

performs slightly better than TRADE for the latter. This is caused by the fact that *SSCA2*'s transactions are smaller than *Intruder*'s, which means that *SSCA2*'s serialization time is shorter.

For applications with limited number of transactions (*Yada*, *Labyrinth* and *Bayes*), the serialization overhead at commit phase is negligible. In these scenarios, establishing relaxed happens-before edges penalizes TRADE with respect to s-TRADE. This effect can be seen especially with *Labyrinth* and *Yada*, while for *Bayes* the two tools perform similarly. More in detail, for *Bayes* the ratio between the number of transactional and nontransactional accesses (0.15) is much lower than *Labyrinth* and *Yada* (15.09 and 26.14, respectively). This low ratio indicates that nontransactional operations are the primary factors in determining the overall execution time. On the other hand, transactional accesses are predominant for both *Labyrinth* and *Yada*. However, *Labyrinth* mainly accesses individual addresses transactionally, which only requires vector clocks allocation and initialization. Non-unique transactional accesses, instead, require TRADE to determine the relaxed happens-before edges: The high overhead for *Yada* is caused by the large number of non-unique transactional accesses performed.

We also analyze the overhead introduced by s-TRADE and TRADE over native execution of STAMP applications. In this case, we use Pipeline for s-TRADE and LLT for TRADE. Table 7.3 reports the execution times for the native execution of STAMP applications running with both STM systems. As reported in Section 7.1, running STAMP applications with LLT provides higher speedups than Pipeline. The columns "Instr. Only" present the execution time when running STAMP applications with Pin only intercepting the necessary instructions required for the two algorithms, such as begin/end transactions, read and write accesses, etc. For some applications (like *Kmeans*),

Apps.	*LLT* without SGLA semantics			*Pipeline* with SGLA semantics		
	Base-time	**Instr. Only**	**TRADE**	**Base-time**	**Instr. Only**	**s-TRADE**
Intruder	1.71	63.65	270.95	4.26	103.28	358.36
SSCA2	0.70	12.73	566.86	1.89	15.40	507.21
Kmeans	3.74	176.30	178.75	5.89	213.41	308.34
Vacation	1.82	98.01	437.10	2.96	72.18	518.78
Genome	1.78	74.49	140.78	2.61	59.43	191.21
Yada	0.10	1.65	6.79	0.55	1.82	3.65
Bayes	5.82	36.11	41.23	10.47	69.98	81.58
Labyrinth	16.12	68.04	78.85	31.37	96.15	111.57

Table 7.3: Performance comparison between TRADE running on LLT and s-TRADE running on Pipeline. Time in seconds.

the instrumentation overhead accounts for the largest part of the total overhead when running the race detection tools. The total execution time of running TRADE and s-TRADE is reported in the columns labeled "TRADE" and "s-TRADE", respectively. While the relative slowdown over the native execution for TRADE and s-TRADE is comparable, the absolute execution time for TRADE is generally considerably lower than s-TRADE, even for *Labyrinth* for which, as shown in Figure 7.8, TRADE shows a high relative slowdown with respect to s-TRADE (*Labyrinth* runs 1.41x faster with TRADE).

Summary: These experiments demonstrate that both tools precisely detect transactional data races for STAMP applications. Our performance evaluation also reveals that TRADE performs similarly to s-TRADE despite the overhead introduced by tracking the relaxed happens-before edges. More importantly, TRADE shows faster overall execution time when coupled with high-performance STM systems, such as LLT or ET, which decreases debugging session time and increases programmers' productivity. Debugging tools such as TRADE are fundamental in the multi-core era, where high performance and scalability are of paramount importance.

7.7 Related Work

In lock-based synchronization, there are well-established requirements to detect whether or not a program satisfies a locking discipline. Under these established requirements, data race detection tools have been also intensively studied. We can classify data race detection for lock-based applications into two main categories: dynamic and static analysis. Static race detectors [49, 78] are based on compile-time analysis of the source code to find all potential data races in any possible execution of a program. Dynamic race detectors [137, 158] rely on program instrumentation or hardware support to monitor memory accesses and synchronization operations. Dynamic tools are often based on lockset [137, 158] or on happens-before [44, 141] relation.

Lockset algorithms enforce the locking discipline where every shared variable is protected by some locks. Basically, each shared variable is associated with a lockset that keeps track of all locks held during accesses, and a race is reported when the lockset becomes empty. Happen-before algorithms are based on Lamport's happens-before relation [93], which combines program order and synchronization events to establish a partial temporal ordering of instructions.

There are only a few works on race detection that leverage TM. Gupta et al. [60] present a system that modifies a HTM implementation to perform dynamic race detection (RaceTM). The authors introduce the concept of lightweight debug transactions that span nontransactional code and exploit the conflict detection mechanisms of TM to detect transactional data races. RaceTM introduces a lower instrumentation overhead than TRADE, as shared memory accesses are tracked by the hardware. On the other hand, RaceTM is sensitive to thread migration and introduces a high number of false positives caused by false sharing inside cache lines, and false negatives caused by cache eviction. TRADE is not affected by many of these issues and does not require extra hardware. Finally, TRADE is based on a formal definition of transactional data races.

Teixeira et al. [154] detect data races in TM applications by converting transactions into lock-protected critical sections and applying an existing lock-oriented data race detector. The authors implemented their approach with AJEX [41], an extension to the Polyglot compiler framework for Java, to parse atomic blocks and to use JChord [118] lock-based data race detector. Elmas et al. [48] present Goldilocks which

is a lockset-based algorithm for precisely computing the happens-before relation and detecting data-races at runtime. The authors implemented the algorithm in the Kaffe Java Virtual Machine and evaluated their system by using Java benchmarks and a few microbenchmarks that combine lock-based and transaction-based synchronization. For the implementation of transactions, the authors use the source-to-source translation and protect all shared objects accessed in a transaction with per-object lock. The problem with these two approaches is that directly transforming transactions into a single global lock serializes the execution of transactions and modifies the run-time characteristics of the application, enforcing an artificial total sequential order.

Unlike previous transactional race detection algorithms, s-TRADE and TRADE do not require serial execution of transactions. Moreover, TRADE does not assume the strict synchronized-with relations and can be used with high-performance STMs. Finally, all the transactions-aware race detection tools proposed in the literature have been evaluated with simple Java micro-benchmarks and cannot be used in production systems. We evaluate s-TRADE and TRADE with the state-of-the-art STM systems and with real and complex C/C++ TM applications, such as STAMP benchmarks.

7.8 Conclusions

TM has reached maturity level with many hardware (HTM) and software (STM) implementations available to programmers. However, there is still lack of debugging tools that automatically check the correctness of TM programs, especially those written with unmanaged programming languages, such as C/C++.

In this chapter we proposed TRADE, a novel algorithm that precisely detects transactional data races for C/C++ TM applications. Our algorithm is based on the relaxed transactional happens-before relation that only orders conflicting transactions. This definition is closer to transactional memory semantics and more intuitive than previously defined strict transactional happens-before relation.

We implemented a dynamic race detection tool based on TRADE: Our tool precisely detects transactional races (real or injected) for STAMP applications. Thanks to our tool, we were able to identify data races in *SSCA2* that have not been previously reported. We also compared TRADE tool with an equivalent race detection tool that implements an algorithm based on the strict happens-before relation (s-TRADE).

TRADE tool shows negligible overhead over s-TRADE but can be used with popular, high-performance STMs, which increases its practicability.

Chapter 8

T-Rex: A Dynamic Race Detection Tool for C/C++ Transactional Memory Applications

8.1 Introduction

The correctness models mentioned in the previous chapter pose some restrictions on the programmer, the language, and the underlying TM implementation. For example, Static Separation (SS) poses restrictions on how memory locations can be accessed during the execution of a program — the same location cannot be written both transactionally and nontransactionally — and does not permit common programming techniques, such as initializing shared memory locations before the main thread creates secondary threads. Dynamic Separation (DS) overcomes some of the limitations of SS by providing explicit operations, invoked by the programmer, to indicate when a location changes from being available for use inside transactions to being available for use directly. Both SS and DS require language support and extensions available only in STM-Haskell and AME, respectively, which limits their applicability. SGLA requires a total ordering among all transactions in the system [110]. However, programmers have been reluctant to use STMs that implements SGLA semantics because enforcing total ordering serializes transactions with respect to their commit order, which considerably

limits performance and scalability, as shown in Figure 7.2. Consequently, most of the high performance STMs do not implement SGLA, which diminishes the applicability of the race detection tool based on *strict transactional happens-before*. The *relaxed transactional happens-before* relation overcomes some of the limitations of SGLA by allowing a higher level of concurrency. However, a race detection algorithm based on the relaxed transactional happens-before relation, such as TRADE, still suffers from the problems of any tool based on a form of happens-before relation: it is costly in terms of performance, sensitive to compiler optimization, and highly depending on thread interleaving.

The definition of transactional data race is central to all correctness models for TM applications as a correct TM program never features transactional data races. In this work, we propose a new definition of transactional data race based on the intuitive notion of data races occurring between accesses to a memory location from different threads, where at least one access is a write and at least one access is non-transactional. This definition is transparent to the underlying STM implementation, can be used for a broad set of C/C++ TM programs, enables a wide range of implementation techniques to be used, and allows the implementation of efficient dynamic race detection tools for TM applications. We only rely on properties that are common to a large number of TM implementations or intrinsic to transactional memory (such as weak isolation) and do not assume other properties that are not widely available. Moreover, our definition is agnostic to thread interleaving, which implies that a transactional data race exists irrespective to the order in which the two threads are scheduled in a particular execution, even if the race does not manifest itself in a particular execution.

Based on this definition, we propose *T-Rex*, a dynamic race detection tool that detects transactional data races for C/C++ TM programs. *T-Rex* records transactional and non-transactional accesses to shared memory locations into per-thread meta-data structures and then detects transactional data races at global synchronization points (such as barriers and application termination). To help programmer resolve bugs, *T-Rex* reports the instruction and memory location addresses and the type of each race.

We evaluated *T-Rex* with a widely-used STM system, TL2 [43], running applications from the STAMP benchmark suite [22] on an 8-core Intel Nehalem server. Our experiments reveal new data races for several STAMP applications. To validate the accuracy of our tool, even for applications that do not present transactional data races,

we inject synthetic bugs and verify that the injected bugs are detected. We perform a detailed performance analysis and provide the overhead breakdown of *T-Rex* when running STAMP applications and identify major bottlenecks. We then show how optimization techniques, such as zero-copy commit phase, effectively reduce these bottlenecks. Finally, our experiments show that *T-Rex*is considerably faster than the race detection tool presented in Chapter 7, TRADE (5.58x faster on average).

This work makes the following contributions:

- We propose a definition of a transactional data race for C/C++ TM programs that does not impose any constraints on STMs.
- Based on this definition, we implement *T-Rex*, a dynamic race detection tool for C/C++ TM applications that provides full coverage with higher performance with respect to previous race detection tools.
- We discover new data races for STAMP applications (*Intruder* and *Bayes*) that have not been previously reported.

This chapter is organized as follows: Section 8.2 motivates our work from different aspects. Section 8.3 introduces our transactional data race definition and its correctness implications; Section 8.4 describes the design, implementation and optimization of *T-Rex*; Section 8.5 provides detailed coverage and performance analysis of *T-Rex*; Section 8.6 concludes this work.

8.2 Motivation

While happens-before algorithms (such as s-TRADE and TRADE) handle many styles of synchronization, they come at a cost. First, tools based on happens-before are usually costly in terms of performance, as they need to access global information and check for the occurrence of data races at every memory access [137, 164]. Second, happens-before algorithms are sensitive to compiler and hardware instruction reordering and optimizations. Third, the effectiveness of these tools is highly dependent on the thread interleaving produced by the scheduler. Consider the example in Figure 8.1: Intuitively, the example includes a data race on `x`, as `Thread 2` accesses `x` without the proper synchronization. However, if `Thread 2` fully executes before `Thread 1` then

Thread 1	Thread 2
atomic {	r1 = x
x = 1;	atomic {
y = x + 1;	y = 42;
}	}

Figure 8.1: This program is intuitively racy but a race detection tool based on relaxed transactional data race definition produces different results according to thread interleaving: if `Thread 2` fully executes before `Thread 1` then the tool does not detect any transactional data races in a given execution.

`Thread 2`'s transaction (T2) is ordered before `Thread 1`'s transaction (T1), i.e., T2 $\longrightarrow_{rhb}$ T1. (`r1 = x`) $\longrightarrow_{rhb}$ T2 by program order, hence (`r1 = x`) $\longrightarrow_{rhb}$ T1, thus TRADE does not detect any transactional data races. Let us now consider a thread interleaving in which `Thread 1` executes before `Thread 2`: in this case T1 $\longrightarrow_{rhb}$ T2 and, by program order, (`r1 = x`) $\longrightarrow_{rhb}$ T2 but we cannot conclude that (`r1 = x`) $\longrightarrow_{rhb}$ T1, thus there is a relaxed transactional data race and TRADE detects a race.

Ideally, a program should be correctly synchronized under a given definition irrespectively of the order in which the two threads are scheduled in a particular execution. Moreover, the definition of relaxed transactional data race does not always follow the programmer's intuition of the execution of a program: For example, the program in Figure 8.1 is intuitively incorrect but a tool based on relaxed transactional happens-before may report that the program is correct.

Privatization and publication are techniques used by programmers to directly access shared objects that are temporarily private to a thread. The code in Figure 8.2 shows a typical privatization example: this program appears to be correct as `shared` is always accessed within a transaction and access to `x` from `Thread 2` is conditioned to the value of `shared`. In this example, either `Thread 1`'s transaction commits first, in which case `Thread 2` never accesses `x` and the final state is `x = 1`, or `Thread 2`'s transaction commits first, in which case the final value is `x = 43`.

Strict and relaxed transactional data race definitions assume support for safe privatization/publication in the STM and recognize these idioms. However, most STM systems do not provide such support due to performance or design complexity reasons [110]: In order to support privatization and publication, the STM must enforce a

Thread 1	Thread 2
`atomic {`	`atomic {`
`  shared = false;`	`  if (shared)`
`}`	`    x = 42;`
`x++;`	`}`

Figure 8.2: Initially `shared = true` and `x = 0`. This program is intuitively correct but may result in incorrect behavior, depending on the underlying STM implementation.

memory barrier before a thread attempts to read a shared memory location. This introduces additional overhead and completely defeats the purpose of accessing a shared variable privately, i.e., shared memory access without the overhead of synchronization. If the underlying STM does not support safe privatization/publication, there might be subtle transactional data races introduced by the STM itself, because of speculative reads, buffered writes or the abort mechanism.[1] We refer to this kind of transactional data races as "STM-centric" data races. Let us consider again the example in Figure 8.2, with a lazy-update/lazy conflict detection STM that does not support safe privatization. Assume that `Thread 2`'s transaction executes first, then `shared` is `true` and `Thread 2` writes `42` to `x`. However, since the STM is lazy update, the writes to the memory are delayed at commit phase. If `x++` in `Thread 1` is performed before the STM writes back to `x` in `Thread 2`, then `x` will be overwritten with the value written by `Thread 2`'s transaction. The final result will be `x = 42`, which is a value "out of thin air".[2]

Compiler and hardware instruction reordering and optimizations can also introduce speculative reads or delayed writes. As discussed in the previous Chapter 7, in the example in Figure 7.1 a compiler could reorder the instructions in `Thread 2` execution and speculatively read `data` before reading `ready`. While both buffered and in-place update STMs suffer from these problems, compiler and hardware optimizations are essential to achieve high performance and cannot simply be disabled.

TRADE does not consider possible STM-centric data races nor compiler/hardware

[1] As explained in Section 7.2, if the underlying STM does not implement privatization-safety, there is an additional constraint on the programmer to guarantee the correctness of privatization and publication idioms.

[2] In eager-update STMs, there are similar problems caused by the fact that a transaction may continue to execute as zombie transaction with its modification visible to other threads [1].

optimizations, thus programs such as the ones described in Figures 7.1 and 8.2, can result in incorrect executions or program crashes even if they appear to be correct according to the relaxed transactional data race definition. In this work we do not rely on any particular STM implementation and opt for an approach in which we do not assume safe privatization/publication support. At the same time, we allow common compiler and hardware optimizations, such as instruction re-ordering.

8.3 Preliminaries

As shown in the previous section, correctness models based on happens-before relation come with some limitations and pose constraints on the STM implementation such as support for safe privatization/publication. Because of performance and implementation reasons, many STMs do not provide such support. For example, TL2 supports neither SGLA nor safe privatization/publication, hence it is an "inconsistent" implementation for strict and relaxed happens-before transactional data race definition [38, 110].

In this section we look for a definition of transactional data race that follows the programmer's intuition of being correctly synchronized while, at the same time, allowing the design and implementation of efficient data race detection tools for a broad set of TM programs and STM implementations. We define as *transactional* an access to a shared memory location that is enclosed by `atomic{...}`. Conversely, we define as *non-transactional* an access to a memory location not enclosed by `atomic{...}`. Let us define W_i^{nt} and R_i^{nt} as the sets of memory locations written and read by thread T_i outside transactions, respectively, and W_i^t and R_i^t as the sets of locations written or read by thread T_i within transactions, respectively. We also define $S_i = \{nt, t\}$ the transactional state of thread T_i: if T_i execution is within a transaction, then $S_i = t$, otherwise $S_i = nt$. We can define the state of a thread T_i at any given moment as $\sigma_i =< R_i^{nt}, W_i^{nt}, R_i^t, W_i^t, S_i >$. The initial state of a thread T_i is $\sigma_i^0 =< \oslash, \oslash, \oslash, \oslash, nt >$. The state of a TM multi-threaded program σ at any given moment is the union of the N threads' states at that moment, i.e., $\sigma = \bigcup_{i=1}^{N} \sigma_i$.

During the program execution, each thread performs several operations that change its state, hence, the program state.

- $rd(i, x)$ and $wr(i, x)$, which read and write a value from x nontransactionally. As a result of $rd(i, x)$, $R_i^{nt} = R_i^{nt} \cup \{x\}$; similarly a $wr(i, x)$ produces a state in which

$W_i^{nt} = W_i^{nt} \cup \{x\}$.

- *txrd*(i, x) and *txwr*(i, x), which read and write a value x transactionally. A *txrd*(i, x) produces a new state in which $R_i^t = R_i^t \cup \{x\}$; the result of a *txwr*(i, x) is $W_i^t = W_i^t \cup \{x\}$.

- *begin*(i) and *end*(i). *begin*(i) starts a transaction and sets $S_i = t$; *end*(i) terminates a transaction and sets $S_i = nt$.[1]

- *barrier*, which blocks a thread T_i until all threads T_j, $j \neq i$, reach the barrier. A *barrier* operation transforms a program state $\sigma = \{\sigma_1, ..., \sigma_N\}$ into an empty state $\sigma' = \{\oslash, ..., \oslash\}$.

We denote the sequence of operations performed by a thread T_i during its execution with α_i. An execution trace α is a sequence of operations performed by all threads in a multi-threaded program that change the state of a program from σ to σ', i.e., $\sigma \Rightarrow^{\alpha} \sigma'$. A particular execution trace α is the combination of the specific interleaving of the sequences of actions of each thread α_i in that execution. It follows that the same sequences of actions $\alpha_1, ...\alpha_N$ can produce different execution traces, one for each possible thread interleaving. If $\sigma \Rightarrow^{\alpha} \sigma'$ and $\sigma \Rightarrow^{\alpha'} \sigma'$, than α and α' are *equivalent* ($\alpha \equiv \alpha'$) and α' can be obtained from α by applying a different thread interleaving. If $\sigma \Rightarrow^{\alpha'} \sigma''$, then α and α' are not equivalent ($\alpha \not\equiv \alpha'$), i.e., there is at least one thread execution trace that is different in the two program execution traces α and α'.

Having defined the possible operations that threads can perform and that affect their status, the simplest definition of transactional data race is the definition of conflict.

Definition 7. Given a program execution α, two memory accesses a and b *conflict* if they access the same location, at least one is a write, and they are executed by different threads.

However, this definition does not take into account that not all conflicts are harmful.

Definition 8. A *benign conflict* between two accesses a and b is a conflict that does not generate incorrect results or crashes in any execution of the program.

[1] We consider a flat model for nested transactions, thus any *begin*(i) after the outmost *begin*(i) will not modify the thread state.

We define a conflict that is not benign as *harmful*. The following conflicting accesses are benign:

- *Transactional accesses.* Accesses to the same shared variable within transactions are benign conflicts. The TM system will take care of detecting the conflict and aborting one of the conflicting transactions.

- *Single-threaded accesses.* Shared variables accessed during sequential parts of the program do not generate harmful conflicts with other accesses in parallel parts of the program. This paradigm enables common practices such as setting the initial value of global variables before creating secondary threads (*initialization*) or writing the final results after joining all secondary threads (*finalization*) without using transactions. *Initialization* is a programming paradigm that transforms a program state $\sigma \Rightarrow \sigma'$ where $\sigma = \{\sigma_1, \oslash, ..., \oslash\}$ (σ_1 is the state of the main thread) and $\sigma' = \{\oslash, \oslash, ..., \oslash\}$. *Finalization* is a programming idiom that produces a state $\sigma = \{\sigma_1, \sigma_2, ..., \sigma_N\} = \{\oslash, \oslash, ..., \oslash\}$; after the finalization $\sigma_2, ..., \sigma_N$ do not change.

- *Global synchronization.* Global synchronization primitives (such as barriers) are points in the program that have to be reached by all threads before proceeding to the next section. Accesses to shared memory locations across global synchronization primitives do not generate harmful conflicts. Global synchronization primitives behave like *barrier* operation and produce a state $\sigma = \{\oslash, \oslash, ..., \oslash\}$.

We can now provide a formal definition of transactional data race:

Definition 9. A *transactional data race* exists between two accesses a and b in an execution α if a and b conflict, the conflict is not benign, and at least one access is non-transactional.

This definition of transactional data race is independent of thread interleaving and relies on the more intuitive idea that two accesses to a shared object without the proper synchronization would probably result in a data race. Similar to data race detectors based on lockset algorithms, this definition does not need to witness a concurrent access to a shared memory location in particular program execution to report a potential race [137]. This is a safe approach as, although programmers occasionally deliberately allow a data

race when the nondeterminism seems harmless, usually a potential data race is a serious error caused by failure to synchronize properly. More formally, given a program execution trace $\alpha = \{\alpha_1, ..., \alpha_N\}$, where $\alpha_1, ..., \alpha_N$ are the thread execution traces of all threads in the system, if α is race-free, then all program execution traces $\alpha' \equiv \alpha$ are also race-free. Similarly, if a race is observed in a program execution α, then the race is present in all the program executions $\alpha' \equiv \alpha$. For example, in the code in Figure 8.1, $\alpha_1 = begin(1); txrd(1, x); end(1)$ and $\alpha_2 = rd(2, x); begin(2); txwr(2, y); end(2)$. If we detect a transactional data race in the execution $\alpha = \alpha_2; \alpha_1$, we can infer that the same race exists in the execution $\alpha' = \alpha_1; \alpha_2$, even if we have not witnessed the transactional data race in α'. Note that s-TRADE and TRADE are not independent of thread interleaving: tools based on their corresponding transactional data race definition will report that the execution $\alpha = \alpha_2; \alpha_1$ is race-free; however, this does not imply that all program executions $\alpha' \equiv \alpha$ are race-free as well. In fact, the same tool will report a happens-before transactional data race for the execution $\alpha' = \alpha_1; \alpha_2$ ($\alpha' \equiv \alpha$). In some cases, different thread interleavings may produce non-equivalent program executions. For the example in Figure 8.2, if `Thread 1` fully executes first, then the program execution is $\alpha = \alpha_1; \alpha_2$, where $\alpha_1 = begin(1); txwr(1, shared); end(1); rd(1, x); wr(1, x)$ and $\alpha_2 = begin(2); txrd(2, shared); end(2)$. If `Thread 2` fully executes first, then the program execution is $\alpha' = \beta_2; \alpha_1$, where `Thread 2` execution trace is $\beta_2 = begin(2); txrd(2, shared); txwr(2, x); end(2)$. Since α_2 and β_2 are two different thread executions (there is an extra $txwr(2, x)$ operation in the latter), $\alpha \not\equiv \alpha'$. A tool based on our definition of transactional data race will report a data race for α' but not for α.

We can define an algorithm to detect transactional data races in TM programs: a transactional data race occurs between thread T_i and T_j if and only if at least one of the following is true:

$$c_i^1(j) = W_i^{nt} \cap \{R_j^{nt} \cup W_j^{nt} \cup R_j^t \cup W_j^t\} \neq \oslash \quad (8.1)$$

$$c_i^2(j) = R_i^{nt} \cap \{W_j^{nt} \cup W_j^t\} \neq \oslash \quad (8.2)$$

$$c_i^3(j) = W_i^t \cap \{R_j^{nt} \cup W_j^{nt}\} \neq \oslash \quad (8.3)$$

$$c_i^4(j) = R_i^t \cap \{W_j^{nt}\} \neq \oslash \quad (8.4)$$

We define the set of transactional data races between T_i and T_j as:

$$\mathbb{S}_i(j) = c_i^1(j) \cup c_i^2(j) \cup c_i^3(j) \cup c_i^4(j) \tag{8.5}$$

A program is correctly synchronized if and only if, $\forall$ threads T_i, T_j such that $i \neq j$, $\mathbb{S}_i(j) = \emptyset$.

We can also define a correctness model for TM programs:

Definition 10. A TM program is correct if and only if no transactional data races exist in any serializable not-equivalent program execution α.

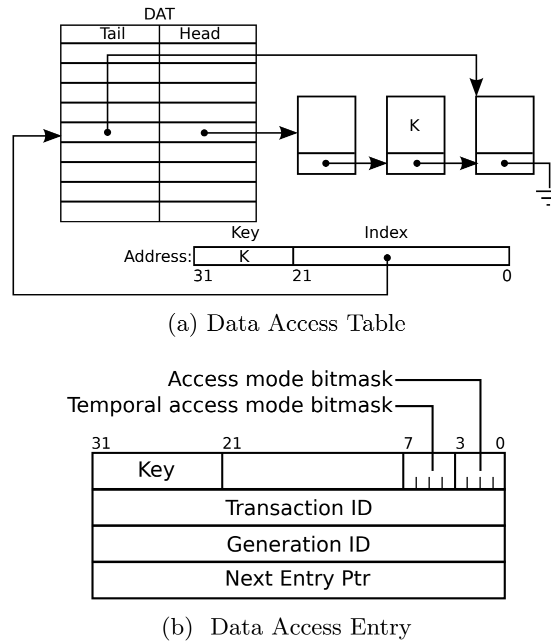
This correctness model is compatible with most of the popular available STM implementations. Moreover, the model allows common programming practices and idioms such as initialization, read-shared and finalization. Finally, a transactional data race detection tool based on this model is less prone to missing transactional data races as compared to a tool based on strict and relaxed transactional happens-before. In fact s-TRADE and TRADE tools need to analyze all possible serializable executions of the program (which requires many repetitions of the same program with the same input), while a tool based on our correctness model only needs to check the not-equivalent executions (which are considerably fewer).

8.4 Design and Implementation

This section presents the design and implementation of *T-Rex*, a dynamic race detection tool for real C/C++ TM applications. *T-Rex* checks if a TM program is correct based on Definition 10. First, we describe the data structures used to record transactional and non-transactional shared memory accesses, then the dynamic binary instrumentation framework used to identify transactions and instrument read/write accesses. Finally, we describe the *T-Rex* transactional data race detection algorithm and several techniques for its optimization.

8.4.1 Threads Data Access Table

To detect possible data races independent of thread interleaving, all individual transactional and non-transactional accesses to shared memory locations and their access modes (read/write, transactional/non-transactional) have to be recorded. *T-Rex* stores



(a) Data Access Table

(b) Data Access Entry

Figure 8.3: *T-Rex* bookkeeping data structures: a) per-thread DAT; b) entry in the per-thread DAT.

each access into a per-thread data access table (DAT), shown in Figure 8.3a. The per-thread DAT is implemented as a hash table and the hash function uses the least significant 22 bits of the memory address (bits [21:0]) to index the table. Addresses that map to the same hash table bucket (aliases) are stored in a linked list, thus all individual accesses are precisely stored without information loss.

Figure 8.3b depicts the structure of a DAT entry. The first word stores the upper part of the memory address (bits [31:22]), used to disambiguate aliases in the same hash table bucket. Bits [3:0] (*access mode bitmask*) store the location access modes: this bitmask is cross checked with the other threads to determine possible transactional data races. The forth and the fifth words store the `head` and `tail` of the list of instructions that have accessed a memory location. The following sections describe the rest of the structure.

8.4.2 Non-Transactional Memory Accesses

To distinguish the transactional accesses from the non-transactional ones, we use the same binary instrumentation framework presented in Chapter 7. At every non-transactional

access, our framework built on top of Pin [100] executes either `PIN_READ()` or `PIN_WRITE()` and inserts/updates the entry corresponding to the memory location in the thread's DAT. `PIN_READ()` and `PIN_WRITE()` set the non-transactional read (bit [1]) and non-transactional write (bit[0]) bit, respectively. Both functions first look for the memory location address into the threads' DAT and, if found, update the access mode bitmask. If the thread has never accessed that particular memory location, a new entry is added and the access mode bitmask properly initialized. For performance reasons, we store both `head` and `tail` of each bucket list: the last element of a bucket list (`tail`) is returned if the required address is not found.

8.4.3 Transactional Memory Accesses

Since our correctness model is transparent to the underlying STM design, *T-Rex* does not rely on the particular STM implementation and data structures (e.g., the read- and write-set) or any other assumption specific to a particular STM. A possible implementation to track transactional read/write accesses consists of using shadow temporal data structures: Once the transaction commits and the memory accesses become permanent, the values in the temporal structures are copied to the thread's DAT. From the performance point of view, however, keeping separate data structures introduce memory copy overhead at commit phase. We, instead, implemented a commit zero-copy algorithm to keep track of transactional accesses. Bits [7:4] in Figure 8.3b store a temporal access mode bitmask used during transaction execution: We use a *transaction ID* to identify memory locations already accessed by the current transaction from those that have never been accessed in the scope of the current transaction. On transactional read/write access, the thread's DAT is searched and, if the memory address is already present in the table, its transaction ID is compared to the current transaction ID. If the entry's transaction ID is smaller than the current transaction ID, this is the first attempt to access that location transactionally and *T-Rex* copies the access mode bitmask (bits [3:0]) to the temporal access mode bitmask (bits [7:4]), updates the access mode bitmask and the entry's transaction ID, and records the DAT entry. From that moment on, every other transactional operation to the same memory location in the scope of the current transaction directly updates the access mode bitmask (bit [3] or bit [2] for `STM_READ()` and `STM_WRITE()`, respectively). If the memory location is not found in the thread's DAT, a new entry is added and its bitmask and transaction

ID are initialized with the current access mode and transaction ID. If the transaction commits, no further update of the thread's DAT is required (zero-copy on commit). On aborts, the access mode bitmasks of the locations accessed during the transaction must be rolled back to their original values. To this extent, *T-Rex* maintain a list of individual memory locations accessed by the current transaction. Moreover, memory locations added by the current transaction (temporal access mode bitmask is 0) are removed from the thread's DAT. Although, with our scheme, aborting is more expensive than committing a transaction, the number of commits is, on average, orders of magnitude higher than the number of aborts, thus we generally have a net gain.

8.4.4 *T-Rex* Race Detection

A thread T_i may have transactional data races with multiple threads at the same time. The following theorems reduce the complexity of computing all the sets of transactional data races.

Theorem 1. *The set of transactional data races that thread T_i has with threads $\{T_j, T_k\}$ is equivalent to the union of the sets of transactional data races:*

$$\mathbb{S}_i(j,k) = \mathbb{S}_i(j) \cup \mathbb{S}_i(k)$$

Proof. This theorem can be proved using the commutativity and distributivity properties of set unions and intersections. □

The equality in Theorem 1 can be also read in the reverse order: if $\mathbb{S}_i(j)$ and $\mathbb{S}_i(k)$ are known, $\mathbb{S}_i(j,k)$ can be obtained by summing the known sets. Since $\mathbb{S}_i(j)$ and $\mathbb{S}_i(k)$ are independent, they can be determined in parallel. Theorem 1 can be generalized to $N \geq 3$:

Corollary 1. *The set of transactional data races that thread T_i has with threads $\{T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_N\}$ is equivalent to the union of the individual sets of races:*

$$\mathbb{S}_i(1, \ldots, i-1, i+1, \ldots, N) = \bigcup_{j \neq i} \mathbb{S}_i(j)$$

Proof. This theorem can be proved using the commutativity and distributivity properties of set unions and intersections. □

This definition still requires computing $\mathbb{S}_i(j)$ for $i, j = 1..N, i \neq j$, where N is the number of threads.

Theorem 2. *The set of transactional data races that thread T_i has with thread T_j is equivalent to the set of transactional data races that thread T_j has with thread T_i:*

$$\mathbb{S}_i(j) = \mathbb{S}_j(i)$$

Proof. The theorem can be proved by applying the commutativity and distributivity properties of set unions and intersection and by properly grouping the terms. □

In our implementation, the sets R_i^t, W_i^t, R_i^{nt}, and W_i^{nt} of a thread T_i are determined by the access mode bitmask in each entry of the thread T_i's DAT (DAT_i). For example, if the entry for the location X stores the access mode bitmask 0110, then $X \in W_i^t$ and $X \in R_i^{nt}$, hence thread T_i can have a transactional data race on X with any thread T_j if $X \in \{W_j^{nt} \cup R_j^{nt}\}$ (8.3) or $X \in \{W_j^{nt} \cup W_j^t\}$ (8.2).

We can express conditions (8.1)-(8.4) introduced in Section 8.3 in terms of bit operations and detect transactional data races through a logic function determined and optimized with Karnaugh maps techniques.[1] Let us define $B_i(X)$ to be the access mode bitmask of thread T_i for location X ($B_i(X) = 0x0$ if $X \notin DAT_i$). Then, for threads T_i and T_j, conditions (8.1)-(8.4) can be expressed as:

$$(8.1) \Rightarrow (B_i(X) \wedge 0x1) \wedge B_j(X) \tag{8.6}$$

$$(8.2) \Rightarrow (B_i(X) \wedge 0x2) \wedge (B_j(X) \wedge 0x5) \tag{8.7}$$

$$(8.3) \Rightarrow (B_i(X) \wedge 0x4) \wedge (B_j(X) \wedge 0x3) \tag{8.8}$$

$$(8.4) \Rightarrow (B_i(X) \wedge 0x8) \wedge (B_j(X) \wedge 0x1) \tag{8.9}$$

for each $X \in DAT_i$ and the number of transactional data races between T_i and T_j on location X can be computed as the sum of the hamming weights of (8.6)-(8.9).

Moreover, by Theorem 1 and Theorem 2, $\forall i, j, k : i \neq j \neq k$, $\mathbb{S}_j(i)$ and $\mathbb{S}_k(i)$ can be independently computed by threads T_j, and T_k and $\mathbb{S}_i(j, k)$ equals to the union of these two sets. $\mathbb{S}_j(i)$ and $\mathbb{S}_k(i)$ can be computed in parallel because each thread's

[1] The representation of Karnaugh maps for 8 variables requires considerable space and we omit it here for the sake of brevity.

DAT is disjoint from the others and *T-Rex* race detection only requires reading the threads' DATs. We, thus, implemented a parallel race detection algorithm in which each thread T_i independently detects transactional races with thread T_j, $\forall j > i$ according to Theorem 2.

As mentioned in Section 8.3, memory accesses across sequential/parallel parts of a program do not incur transactional data races. *T-Rex* keeps track of the number of current active threads and enable bookkeeping only when there are $N \geq 2$ active threads.

Similarly, memory accesses from different threads across global synchronization points are also race-free. When an application reaches a global synchronization point, *T-Rex* performs race detection for the memory locations accessed by the threads between this global synchronization point and the previous one, and then safely discards the entries in the threads' DATs. Instead of deallocating/allocating threads' DATs at every global synchronization point, which is costly, we discard the DATs' entries by invalidating their *generation ID*: after data race detection, *T-Rex* increases the current global generation ID, which invalidates all the previous entries in the DATs. The generation ID of an entry is recorded when the memory location is first inserted into the thread's DAT and it is valid as long as the value is equal to the global current generation ID. As a further memory optimization, invalid entries can be re-used when inserting a new memory location in the current generation. We use the same technique to avoid deallocation of entries inserted into the thread's DAT from within transactions: on abort, instead of deallocating entries, we artificially decrease their generation numbers.

Extensions: Although privatization/publication are not directly considered as benign conflicts, *T-Rex* provides debugging primitives (`begin_private` and `end_private`) to filter out warnings produced by those idioms. These primitives should only be used if the underlying STM provides support for safe privatization/publication and does not introduce STM-centric data races.

Another programming style consists of creating/terminating threads (besides the initialization/finalization techniques described previously) during the program execution. *T-Rex* handles this case by introducing artificial global synchronization points every time a thread forks/terminates a child thread. This essentially shortens the

Appl.	# Transactional Data Races		# Extra Races with bug injection
	Location Addresses	Instruction Addresses	
Intruder	540723	5	7723*
Ssca2	99	7	3
Kmeans	0	0	3546
Vacation	0	0	758354
Genome	0	0	1706
Yada	0	0	6
Bayes	260	7	3
Labyrinth	0	0	7

Table 8.1: Number of detected transactional data races for STAMP applications for the original version and a version with synthetic bugs injected. **Intruder* crashed because of the injected bug.

execution window inside which race detection is applied but introduces a timing dependence, which makes the algorithm sensitive to thread interleaving. In our experience, these cases are rare (none of the tested applications uses this technique). *T-Rex* is not configured to recognize this paradigm by default to avoid timing dependences and preserve the independence of thread interleaving.

8.5 Experimental results

This section evaluates the coverage and performance of *T-Rex* and compares *T-Rex* with TRADE on a Intel Nehalem system (8 cores, 16GB of RAM). We run applications from the STAMP benchmark suite, widely used to test TM systems, on TL2 [43]. STAMP applications and TL2 are compiled with gcc 4.4.5 with optimization `O3` for 64-bit architectures. In all the experiments we run eight parallel threads.

8.5.1 *T-Rex* Race Detection Coverage

This section analyzes transactional data race conditions for STAMP applications and *T-Rex* coverage. Table 8.1 shows the number of transactional data races detected by *T-Rex* (both the number of instructions and memory locations) for all the STAMP applications.

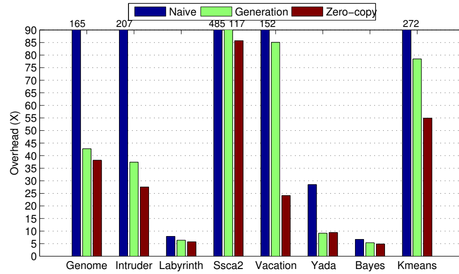
Detected Races: Even though STAMP benchmarks are mature applications, *T-Rex* detects transactional data races for *Intruder*, *Bayes* and *SSCA2*. To the best of our knowledge, this is the first study that reports these transactional data races

and their internal details. Our further investigation shows that the detected data races for *Intruder* and *Bayes* are harmful if the underlying STM does not support safe privatization/publication. For *SSCA2*, the detected races are harmful even if the STM systems support SGLA and safe privatization/publication. Finally, we analyze whether those data races are detected by TRADE.

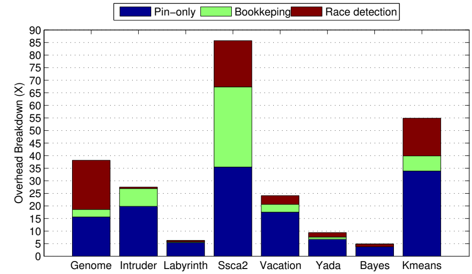
T-Rex reports exactly the same transactional races detected by TRADE for *SSCA2* in the process of creating the inner vertex list. Since the code that generates the inner vertex list is enclosed between two global barriers and no transactions are used to protect accesses to the graph's nodes, the accesses to the nodes are always detected as a race by both *T-Rex* and TRADE.

Intruder is a signature-based network intrusion detection systems (NIDS) application that scans network packets for matches against a known set of intrusion signatures. Incoming packets are stored in a FIFO queue while a self-balancing tree dictionary contains the lists of packets that belong to the same session. Both these data structures are shared among the threads. The application consists of three parallel phases: capture, reassembly, and detection. In the TM implementation, the capture and reassembly phases are enclosed inside transactions and populate the FIFO queue and the dictionary. In the detection phase, threads retrieve packets from the FIFO queue and detect possible intrusions. Although both the FIFO and the packets are shared data structures, once a packet has been retrieved from the FIFO, no other thread is allowed to work on that packet anymore (privatization), thus there is no need to enclose the detection phase inside transactions. However, each packet is modified by both transactional (capture and reassembly) and non-transactional (detection) code, thus *T-Rex* reports transactional data races. No other transactional data races are reported by *T-Rex* besides the ones just described.

Bayes implements an algorithm for learning Bayesian networks from observed data using a hill-climbing strategy that uses both local and global search. At each iteration, a thread receives a variable to analyze and updates the network with new dependencies. In the transactional version, transactions are used to extract a variable from the task list (privatization) and to add the variable back to the list (publication). After extracting a variable, a thread need not protect its modifications through transactions. This program is racy because several threads extract the same variable from the list and



(a) *T-Rex* runtime overhead.



(b) *T-Rex* runtime overhead breakdown for Zero-copy.

Figure 8.4: *T-Rex* overall overhead and overhead breakdown for STAMP applications.

then access the variable directly without using a barrier to delimit the two portions of the code.

s-TRADE and TRADE do not detect any those STM-centric transactional races for *Intruder* and *Bayes* because they establish happen-before relations between the transaction that privatizatizes the data and the prior transactions while assuming that safe privatization is guaranteed by the underlying STM system. Since most of the commonly-used STMs do not support safe privatization and publication, we believe that it is crucial to discover these STM-centric transactional data races. These races are subtle and likely to be bugs in practice with some typical STM systems. More importantly, it is very difficult for the programmer to reason about incorrect results in the absence of transactional data races.

Injected Races: Similarly to the previous chapter, we inject bugs in the STAMP applications in the form of removing transactions, which transforms transactional sections into non-transactional ones in order to verify that *T-Rex* detects transactional data races without any miss. The third column in Table 8.1 shows the number of transactional data races detected by *T-Rex* when injecting bugs in the STAMP applications. *T-Rex* detects all the transactional data races produced by the injected bugs. If these bugs results in an application crash (malign faults), we detect data races before the application crash (*T-Rex* intercepts the `SIG_KILL`/`SIG_ABORT` signals).

Appl.	TXs	Syn.	#Transactional Accesses		#Nontransactional Accesses	
			Rd	Wr	Rd	Wr
Intruder	6,045K	1	55,752K	3,164K	25,699K	10,338K
Ssca2	5,558K	47	2,780K	5,560K	157,486K	34,243K
Kmeans	10,207K	302	13,113K	6,588K	513,365K	1,507K
Vacation	2,097K	1	288,957K	7,099K	37,639K	14,645K
Genome	2,489K	258	58,288K	1,638K	10,392K	6,259K
Yada	30K	1	1,647K	240K	68K	57K
Bayes	2K	4	32K	3K	218K	19K
Labyrinth	1K	1	92K	91K	4K	3K

Table 8.2: STAMP applications' characteristics.

8.5.2 Overhead analysis

Figure 8.4a shows the *T-Rex* runtime overhead over the native execution of STAMP applications running with TL2. We report the overhead of three different implementations: *Naïve* uses our race detection algorithm but deallocates/allocates threads' DATs at each global synchronization point, after performing a race detection. This version also uses shadow data structures to store temporal transactional read and write accesses that are copied back to the thread's DAT after successful commits. Temporal data structures are allocated at the beginning of a transaction and deallocated at commit phase and on abort. *Generation* employs generation across global synchronization points (no need to deallocate/allocate threads' DATs) but still uses shadow data structures for transactional accesses. To the contrary of the previous case, temporal data structures are not deallocated at the end of transactions but invalidated through a specific generation ID. Finally, *Zero-copy* uses both our optimizations: generation across global synchronization points and zero-copy commit phase.

By comparing the versions *Naïve*, *Generation* and *Zero-copy* we can perceive the effects of each optimization. The use of generations across the global synchronization points considerably improve the performance for the applications (such as *Genome*, *Kmeans* and *SSCA2*) that use barriers or fork/join (see Table 8.2). Moreover, version *Generation* does not deallocate/allocate temporal data structures for transactional accesses, hence applications with a large number of commits (*Intruder*, *SSCA2* and *Kmeans*) also report large performance improvements. Including our zero-copy on commit optimization improves performance for all applications. Our zero-copy on commit technique removes the memory copy overhead of moving transactional accesses

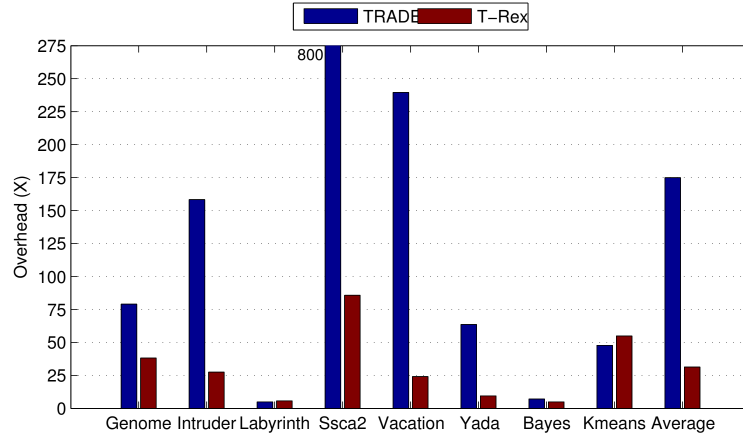


Figure 8.5: Comparing TRADE and *T-Rex* execution overhead over native execution with TL2.

from the temporal data structure to the threads' DATs. The overhead of applications with a large number of commits (*Kmeans*, *Intruder* and *SSCA2*) or large read- and write-sets (*Vacation*) considerably reduces.

For *SSCA2* and *Kmeans*, the runtime overhead is particularly high: we analyzed the overhead breakdown (Figure 8.4b) and discovered that, for these benchmarks, the pure Pin instrumentation overhead (no bookkeeping and no race detection) is dominant. This overhead is 35.5x and 33.9x for *SSCA2* and *Kmeans*, respectively. We further examined the reasons why we experience large instrumentation overhead for some applications such as *SSCA2* and *Kmeans*: Table 8.2 reports the number of commits, synchronization points, and transactional/non-transactional read and write accesses. Note that Pin instrumentation overhead is higher for applications with a large number of read/write accesses and transactions.

From Figure 8.4b and Table 8.2, we can also derive conclusions about the bookkeeping, the read and write instrumentation and the race detection overheads. In the graph, bookkeeping also includes the overhead of tracking read/write accesses through Pin. As expected, applications with large numbers of memory accesses (*SSCA2* and *Vacation*) show larger bookkeeping overhead. However, this overhead is larger for applications with a high percentage of unique accesses. For example, *SSCA2* (55% of unique accesses) shows considerably higher bookkeeping overhead than *Vacation* (2%

of unique accesses). Finally, Figure 8.4b shows that the pure overhead of our race detection algorithm is marginal compared to instrumentation and bookkeeping. Only *Genome*, *SSCA2* and *Kmeans* show noticeable overhead. These applications are the only ones that frequently use global synchronization, thus *T-Rex* performs transactional race detection several times.

Figure 8.5 shows the execution overhead over native execution of TRADE compared to *T-Rex*. Even though TL2 is an inconsistent implementation for TRADE because it does not support safe privatization/publication (the programmer must ensure correctness of these idioms), we run both race detection tools on the same STM to be able to provide fair comparison. As we can see from the graph, the overhead of *T-Rex* and TRADE is comparable for *Labyrinth*, *Bayes*, and *Kmeans*. On the other hand, *T-Rex* overhead is much lower than the one introduced by TRADE for those applications that present large numbers of transactions, such as *Genome*, *Intruder*, *SSCA2*, and *Vacation*, up to 800x over the native execution for *SSCA2*. This is mainly caused by the difficulty of tracking the happens-before relations between all the accesses to shared memory locations. In particular, the vector clocks used to track happens-before relations are shared data structure (as opposed to *T-Rex* DATs) that need to be protected by lock and limit scalability (i.e., the overhead is larger when the thread count is higher).

8.6 Conclusions

Despite the level of maturity reached by transactional memory and the many implementations available at both hardware and software levels, there is still a lack of consensus on the notion of what it means for a TM program to have a data race. Previous correctness models and definition of transactional data races come with some limitations and impose restrictions on the STM implementations: for example, s-TRADE requires the underlying STM to support total ordering among all transactions in the system, which limits the applicability of s-TRADE to most of the STM designs. To overcome these limitations, we propose a new definition of transactional data race that is more intuitive, does not constrain the underlying STM implementation, is independent of thread interleaving but, at the same time, allows common programming idioms and practices.

Based on this definition, we implement *T-Rex*, a precise dynamic checking tool for C/C++ TM applications. *T-Rex* is able to efficiently detect transactional data races in complex, real programs, such as STAMP applications. We discovered transactional data races in some STAMP programs that, to the best of our knowledge, had not been previously reported. Our results show that *T-Rex* is considerably faster than a race detection tool based on TRADE (5.58x on average).

Part V

Conclusions

Chapter 9

Conclusions

Increasing performance through higher processor frequency has reached a sudden stop caused by three major technical bottlenecks: 1) the increasing gap between processor and memory speed (*Memory Wall*), 2) the increasing difficulty of finding enough parallelism within a single stream of instruction to keep the processor utilization high (*ILP Wall*), and 3) the increasing power consumption, which grows exponentially with the processor operation frequency (*Power Wall*). This combination of factors has motivated the major processor manufacturers to shift towards a processor design that includes several computing elements (cores or hardware threads) within the same processor die.

Chip Multithreading (CMT) processors promise to deliver higher performance by running more than one stream of instructions in parallel rather than by increasing the processor's frequency. CMT processors come with different architectures: Chip Multi-Processor (CMP), Simultaneous Multi-Threading (SMT), or a combination of them. Market trends show that the core count has been consistently increasing over the last years and chips with 8 cores and 32/64 hardware threads are commonly available. Moreover, CMT processors deliver higher performance/watt ratios [162] than single thread architectures, which makes them suitable for power-constrained systems, such as data centers. In order to exploit CMT's capabilities, programmers have to parallelize their applications. However, efficiently parallelizing applications, especially at large scale, is difficult and prone to errors and race conditions, such as dead and live locks. Several proposals focus on how to reduce the effort of parallelizing applications on CMT machines. Novel shared memory programming models, such as OpenMP [125],

PGAS [122, 157], Charm++[80], and Transactional Memory [72], that have the potentiality to simplify parallel programming and to enable users to extract higher level of parallelism are seeing wider use in various fields, including high performance computing, data centers and server markets.

Transactional Memory (TM) [46, 66, 72] is a promising programming model that addresses ease of programmability of parallel programs while keeping up with performance expectations of multi-core processors. Transactional Memory allows programmers to mark compound statements in parallel programs with the expectation that the underlying run-time implementation will execute such transactions concurrently whenever possible, generally by means of speculation – optimistic but checked execution, with rollback and retry when conflicts arise. The principal goal of TM is to simplify synchronization by raising the level of abstraction, breaking the connection between semantic atomicity and the means by which that atomicity is achieved. Secondarily, TM has the potential to improve performance, most notably when the practical alternative is coarse-grain locking.

Researchers have proposed different designs and implementations of transactional memory system, from Hardware Transactional Memory (HTM) to Software Transactional Memory (STM) and to hybrid solutions. Until recently, however, most of these solutions mainly focused on implementations that respect TM semantics and the ACI properties. While this is a necessary step, it produced a multitude of HTM and STM designs and benchmark suites that are usually not compatible with each other. Moreover, programmers have been reluctant to use TM because of the generally limited performance and scalability and because of the lack of development tools (e.g., compilers, debuggers, race detectors) and common benchmarks to evaluate which TM design best suits their needs.

More recently, there has been significant effort to consolidate research designs and implementation into industrial standards for transactional memory. This work is summarized in Draft Specification of Transactional Language Constructs for C++ [6]. Moreover, the availability of mature compilers, both from the open source community (GCC-TM [138]) and from the industry (Intel [35, 161], Microsoft [68]), and hardware transactional memory implementations, IBM BG/Q [63] and Intel Haswell[132], show that transactional memory has now reached a maturity level and that it can be used in the production environments rather than just the research environments.

However, in order for transactional memory to be widely adopted in mainstream parallel programming, there is need of filling the gap between the research prototypes developed in the last years and the industry-level standards and products. This dissertation presents work towards improving the practicality of transactional memory across three dimensions. Specifically, this dissertation makes the following contributions:

- **Comprehensive evaluation of TM systems**

  We developed RMS-TM, a comprehensive TM benchmark suit that includes realistic applications from the Recognition, Mining, and Synthesis (RMS) domain. RMS-TM addresses ongoing TM research issues (e.g., nested transactions, I/O operations, system and library calls inside transactions) and provides the potential for straightforward comparison against locks. We also developed a library interface that allows a reliable and fair performance comparison of TM proposals developed by different research groups. Our interchangeable software layer makes it possible to interchange the STM library while keeping the benchmarks source and binary code unaltered.

- **Design and implementation of a high-performance STM**

  We have designed a novel parallel STM implementation, *Software Transactional Memory for Simultaneous Multi-threading* systems — STM^2. STM^2 reduces the runtime overheads by offloading the time-consuming TM operations to a *auxiliary thread* running on *sibling* core/hardware thread. *Application threads* optimistically perform their computation with minimal support from the underlying STM system. All synchronization and STM management operations are performed by the paired auxiliary threads. We exploit the fact that, on modern multi-core processors, sets of cores can share L1 or L2 caches. This lets us achieve closer coupling between the application thread and the auxiliary thread (when compared with a traditional multi-processor systems). We show that our approach outperforms several well-known STM implementations for various TM applications, with average speedups between 1.8x and 5.2x over the tested STM systems. We further enhance STM^2's performance by effectively partitioning processor resources between application and auxiliary threads. In order to bias the allocation of hardware resources in favor of the most demanding thread, we leverage the

hardware thread prioritization mechanism implemented in POWER machines. Results show that effective hardware resource partitioning performs, in general, better than the original STM^2, up to 86% performance improvement.

- **Providing Correctness Semantics for TM applications**

  We propose two novel and precise race detection algorithms and tools for TM applications. The first tool, namely TRADE, is based on a relaxed definition of the happens-before relation and removes the SGLA constraints on the underlying STM system. This algorithm can be used with a broader set of high-performance, scalable TM systems but still requires support for safe-privatization. The second algorithm *T-Rex*removes this constraint. *T-Rex* is based on a new definition of transactional data race that follows the programmers intuition of racy accesses, is independent of thread interleaving, can accommodate popular STM designs, and allows common programming idioms. We implement the race detection tools for C/C++ TM applications corresponding to our algorithms (TRADE and *T-Rex*). For comparison reasons, we also implemented a race detector based on the strict happens-before relation. We compare precision and run-time overheads of our race detection tools. We also analyzed the implications of each semantics on parallel programming and STM implementations. Our experiments show that our tools precisely detect transactional data races. However, *T-Rex* is considerably faster than the tools based on (strict or relaxed) happens-before relation and can be used with a broader set of scalable STM designs because it does not pose implementation constraints.

In summary, this dissertation presents novel techniques and important findings towards improving the efficiency and practicality of TM systems, especially when running on massive multithreaded systems. Both TM system designers and application developers can use the techniques and findings described in this dissertation.

As future work, we plan to analyze the interaction of TM with other programming models and synchronization primitives within the same applications. Such scenarios can arise, for example, when using distributed shared memory systems where application threads can use TM to synchronize access to a shared memory location within a single node and message passing (e.g., MPI) when communicating across different nodes. Systems such as BG/Q are ideal testbeds for such scenarios. A second scenario involves

the use of transactions and lock in legacy applications that are gradually ported to TM. In both cases, the semantics of what it means for a program to be correctly synchronized and our race detection algorithms need to take into account the interaction between TM and the other programming models used. Orthogonally, we plan to extend some of the techniques proposed in this dissertation, e.g., assisted execution, to other systems, such as OS exception handlers [166], dynamic check in Java Script [109] or our race detection algorithms.

Appendix A

TRADE correctness proofs

In this appendix, we provide the logical reasoning of the algorithms' correctness proofs. For brevity, we prove the correctness of TRADE; the proof for s-TRADE is a simplified version of this.

We begin the correctness proofs with the formal definition of program state σ. At any given moment a program is in a state σ represented by a tuple $< C, R, W, TR, TW >$.

Definition 11. A state $\sigma =< C, R, W, TR, TW >$ is *well-formed* if and only if:

1. for all $t, u \in Tid$, if $t \neq u$ then $C_u(t) < C_t(t)$
2. for all $x \in Var$, $t \in Tid$, $R_x(t) \leq C_t(t)$
3. for all $x \in Var$, $t \in Tid$, $W_x(t) \leq C_t(t)$
4. for all $x \in Var$, $t \in Tid$, $TR_x(t) < C_t(t)$
5. for all $x \in Var$, $t \in Tid$, $TW_x(t) < C_t(t)$

Let σ_0 be the initial state where all vector clocks in the tuple are empty, then σ_0 is well-formed.

An execution trace α is a sequence of operations performed by all threads in a multi-threaded program. The operations that a thread $t \in Tid$ can perform are the following:

- $rd(t, x)$ and $wr(t, x)$, which read and write a value from x nontransactionally,
- $txrd(t, x)$ and $txwr(t, x)$, which read and write a value x transactionally,

- $begin(t)$ and $end(t)$, which start and end a transaction,
- $fork(t, u)$, which forks a thread u,
- $join(t, u)$, which blocks until a thread u terminates, and
- $barrier$, which blocks until all threads $u \neq t$ reach the barrier.

When performing an operation a, a program moves from an initial state σ to a final state σ'. We indicate this transition with $\Rightarrow^a$. If the operation a is race-free and σ is well-formed then σ' is also well-formed.

Theorem 3. *(Soundness) Given a well-formed state σ and $\sigma \Rightarrow^\alpha \sigma'$, if* TRADE *detects a data race ϕ, then ϕ is a race in α.*

Proof. If TRADE detects a race between two operations a and b, then

i) a and b conflict, and

ii) $a \not\rightarrow_{whb} b$.

Let's assume that a and b do not race in α, then either a and b do not conflict, which contradicts (i), or $a \longrightarrow_{rhb} b$, which contradicts (ii). Hence a races with b in α. □

Lemma 1. *Let σ_a be a well-formed state and $a.\alpha$ be an execution trace such that $\sigma_a \Rightarrow^{a.\alpha} \sigma_b \Rightarrow^b \sigma'_b$. Let's also assume $t = tid(a)$ and $u = tid(b)$. If $C_t^a(t) \leq C_u^b(t)$, then $a \longrightarrow_{a.\alpha.b} b$.*[1]

Theorem 4. *(Completeness) Given a well-formed state σ and $\sigma \Rightarrow^\alpha \sigma'$, all data races in α are detected by* TRADE*.*

Proof. The theorem can be proved by contradiction. Suppose that there is a data race in α not detected by TRADE. This means that there are two operations in α, say a and b, that conflict and $a \not\rightarrow_{whb} b$. The proof can be constructed by induction on $\alpha = a.\beta.b$, assuming that the data race between a and b is the first in the trace and that there are no other races between β and b. The process can then be repeated with $\alpha = \beta$. Finally, let's assume $t = tid(a)$ and $u = tid(b)$; by definition of data race $t \neq u$ and by Lemma 1 $C_t^a(t) > C_u^b(t)$ because $a \not\rightarrow_{whb} b$.

Without loss of generality, let's assume that a is a write operation to $x \in Var$ and b is a read operation from $x \in Var$. C_t^a and C_u^b denote the thread vector clocks at the time the operations a and b are executed.

[1] The proof of this lemma can be found in [54]. We omit it here.

- $a = txwr(t,x)$, $b = txrd(u,x)$. If $begin(u)$ occurs after $end(t)$ then:[1]

  $TW_x^a(t) = C_t^a(t)$ by $txwr(t,x)$

  $C_u^b = C_u \sqcup TW_x^a$ by $txrd(u,x)$

  If follows that $C_t^a(t) = C_u^b(t)$, which contradicts the initial hypothesis $C_t^a(t) > C_u^b(t)$, hence $a \longrightarrow_{rhb} b$.

- all other cases (at least one operation is nontransactional).

  $W_x^a(t) = C_t^a(t)$ by $txwr(t,x)$, or $wr(t,x)$

  $W_x^a \sqsubseteq C_u^b$ is false in the $rd(u,x)$, or $txrd(u,x)$, because $W_x^a(t) > C_u^b(t)$

  It follows that TRADE detects a race between a and b, which contradicts the initial hypothesis that there was a race in α not detected by TRADE.

$\square$

[1]If the two transactions run in parallel and there is a conflict between a and b, the TM system will abort one of the transactions, hence a data race cannot occur, which contradicts the initial hypothesis.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–74, 2008. 159

[2] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 185–196, 2009. 12, 76, 129

[3] M. Abadi, T. Harris, and K. F. Moore. A model of dynamic separation for transactional memory. *Inf. Comput.*, pages 1093–1117, 2010. 125

[4] J. Abeles, L. Brochard, L. Capps, D. DeSota, J. Edwards, B. Elkin, J. Lewars, E. Michel, R. Panda, R. Ravindran, J. Robichaux, S. Kandadai, and S Vemuganti. Performance guide for HPC applications on IBM power 755 system, 2010. 75, 80, 81, 94

[5] A. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 2006. 19

[6] A. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft specification of transactional language constructs for C++, Aug. 2009. Version 1.1. 49, 51, 57, 64, 126, 131, 143, 180

[7] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 248–259, 2000. 3

[8] K. Albayraktaroglu, A. Jaleel, Xue Wu, M. Franklin, B. Jacob, Chau-Wen Tseng, and D. Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 2–9, Washington, DC, USA, 2005. 28

[9] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, pages 483–485, 1967. 69

[10] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2005. 20

[11] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luj'an, and K. Jarvis. Lee-TM: A non-trivial benchmark suite for transactional memory. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing*, pages 196–207, Agia Napa, Cyprus, 2008. 27, 46

[12] L. Baugh and C. Zilles. An analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *Proceedings of International Symposium on Performance Analysis of Systems and Software*, pages 54–62, Washington, DC, USA, 2008. 28

[13] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Toronto, ON, Canada, 2008. 28

[14] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006. 40

[15] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. *The Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2005. 12, 129

[16] Hans-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 15–15, 2009. 126

[17] C. Boneti, F. Cazorla, R. Gioiosa, C-Y. Cher, A. Buyuktosunoglu, and M. Valero. Software-Controlled Priority Characterization of POWER5 Processor. In *Proceedings of the 35th IEEE International Symp. on Computer Architecture*, Beijing, China, June 2008. 92, 95, 96, 101, 108, 109, 119

[18] C. Boneti, R. Gioiosa, F. Cazorla, J. Corbalan, J. Labarta, and Mateo Valero. Balancing HPC applications through smart allocation of resources in MT processors. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symp.*, Miami, FL, 2008. 95, 99, 109, 112, 119

[19] C. Boneti, R. Gioiosa, F. J. Cazorla, and M. Valero. A dynamic scheduler for balancing HPC applications. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008. 95, 99, 109, 119

[20] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development*, (6), 2000. 92

[21] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. 29

[22] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The International Symposium on Workload Characterization*, 2008. 27, 34, 46, 50, 55, 63, 69, 81, 93, 112, 127, 156

[23] D. Carlstrom, B, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–13, 2006. 19

[24] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why is it only a research toy? *ACM Queue*, pages 46–58, 2008. 5, 69, 72, 73, 84

[25] J. Casper, T. Oguntebi, S. Hong, N. G. Bronson, C. Kozyrakis, and K. Olukotun. Hardware acceleration of transactional memory on commodity systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 27–38, 2011. 81, 88

[26] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. QoS for high-performance SMT processors in embedded systems. *IEEE Micro*, (4), 2004. 93

[27] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 97–108, 2007. 20, 27, 29, 33, 34

[28] B. Chapman, G. Jost, and R. Van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. 29

[29] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 186–195, 1999. 87

[30] S. Choi and D. Yeung. Learning-based SMT processor resource distribution via hill-climbing. *SIGARCH Computer Architecture News*, (2), 2006. 93

[31] D. Christie, J. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Dresden TM Compiler (DTMC). In *Proceedings of the 5th ACM European Conference on Computer Systems*, 2010. 125

[32] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. *SIGPLAN Notices*, 41(11):347–358, 2006. 45

[33] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 14–25, 2001. 87

[34] TM compiler support in gcc 4.7. Project web site. http://gcc.gnu.org/gcc-4.7/changes.html. 125, 143

[35] Intel Corporation. In *Intel C++ STM Compiler Prototype Edition 2.0 Language Extensions and Users Guide*, 2008. 19, 180

[36] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, USA, Mar. 2011. 21, 50

[37] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott. Capabilities and limitations of library-based software transactional memory in C++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, 2007. 5, 49

[38] L. Dalessandro and M. L. Scott. Strong isolation is a weak idea. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009. 125, 126, 129, 130, 131, 160

[39] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 67–78, 2010. 4, 5, 22, 50, 69, 81, 119, 125, 126, 131, 132

[40] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, 2006. 4, 21

[41] R. Dias and B. C. Teixeira. Ajex: A source-to-source java stm framework compiler. Technical report, DI-FCT/UNL, 2009. 152

[42] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, 2009. 4, 21, 125

[43] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the International Symposium on Distributed Computing*, pages 194–208, 2006. 4, 17, 19, 21, 22, 50, 81, 119, 125, 126, 131, 132, 156, 170

[44] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 85–96, 1991. 152

[45] J. Dongarra, P. H. Beckman, T. Moore, P. Aerts, G. Aloisio, J. Andre, D. Barkai, J. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. M. Chapman, X. Chi, A. N. Choudhary, S. S. Dosanjh, T. H. Dunning, S. Fiore, A. Geist, B. Gropp, R. G. Harrison, M. Hereld, M. A. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. E. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. M"uller, W. E. Nagel, H. Nakashima, M. E. Papka, D. A. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. L. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. E. Trefethen, M. Valero, A. Steen, J. S. Vetter, P. Williams, and K. A. Wisniewski, R.and Yelick. The international exascale software project roadmap. *IJHPCA*, 25(1):3–60, 2011. 91

[46] Ulrich Drepper. Parallel programming with transactional memory. *ACM Queue*, pages 38–45, 2008. 11, 180

[47] Michel Dubois and Ho Song. Assisted execution. Technical Report GENG-98-25, University of Southern California, Los Angeles, 1998. 70

[48] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 2007. 152

[49] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 9th ACM symposium on Operating systems principles*, pages 237–252. 152

[50] J. Ennals, R. *Adaptive Evaluation of Non-Strict Programs. PhD thesis.* University of Cambridge, 2004. 46

[51] Object-relational main-memory embedded database system. http://sourceforge.net/projects/fastdb/. 12

[52] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, USA, Feb. 2008. 50

[53] P. Felber, T. Riegel, C. Fetzer, M. S"usskraut, U. M"uller, and H. Sturzrehm. Transactifying applications using an open compiler framework. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007. 65

[54] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 121–133, 2009. 186

[55] V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. QuakeTM: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd International Conference on Supercomputing*, 2009. 46

[56] David Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005. 3

[57] B. Gibbs, B. Atyam, F. Berres, B. Blanchard, L. Castillo, P. Coelho, N. Guerin, L. Liu, C. Diniz Maciel, and C. Thirumalai. *Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations*. IBM Redbook, 2005. 109, 119

[58] D. Grossman, J. Manson, and W. Pugh. What do high-level memory models mean for transactions? In *Proceedings of the workshop on Memory system performance and correctness*, pages 62–69, 2006. 126, 130, 131

[59] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A benchmark for software transactional memory. *SIGOPS Operating Systems Review*, 41(3), 2007. 27, 46

[60] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Rotteler. Using hardware transactional memory for data race detection. In *Proceedings 23rd International Parallel and Distributed Processing Symposium*, 2009. 152

[61] H. Q. Le, W. J Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, (6), 2007. 93

[62] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102–113, New York, NY, USA, 2004. 20, 27, 41

[63] Ruud Haring. The Blue Gene/Q compute chip. In *The 23rd Symposium on High Performance Chips (Hot Chips)*, 2011. 4, 125, 180

[64] V. Haris, G. Neelam, and M. S. Michael. Pathological interaction of locks with transactional memory. In *Proceedings of the 3th ACM SIGPLAN Workshop on Transactional Computing*, 2008. 130

[65] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, 2003. 19

[66] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition.* Morgan and Claypool Publishers, 2nd edition, 2010. 11, 180

[67] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005. 19, 125

[68] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 14–25, 2006. 5, 65, 69, 180

[69] M. Herlihy and Y. Lev. TM DB: A generic debugging library for transactional programs. In *Proceedings 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009. 125

[70] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications*, pages 253–262, Portland, OR, USA, 2006. 19, 27

[71] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of ACM Symposium on Principles of Distributed Computing*, 2003. 19

[72] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 289–300, 1993. 4, 11, 20, 180

[73] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization*, 2010. 93, 96

[74] C. Hughes, J. Poe, A. Qouneh, and T. Li. On the (dis)similarity of transactional memory workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, 2009. 28

[75] Intel C++ STM Compiler, Prototype Edition. urlsoftware.intel.com/en-us/articles/intel-c-stm-compiler-prototype-linebreak[1]urledition/. 57

[76] Intel Corporation. Intel AtomTM processor n450, d410 and d510 for embedded applications, 2010. Document Number: 323439-001 EN, revision 1.0. 93

[77] Intel Corporation. Intel transactional memory compiler and runtime application binary interface, May 2009. Document No. 318523-002US, revision 1.1. 50, 125

[78] Choi J., A. Loginov, and Sarkar V. Static datarace analysis for multithreaded object-oriented programs. Technical report, IBM Research Division, Thomas J. Watson Research Centre, 2001. 152

[79] M. V. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proceedings of the 12th International Parallel Processing Symposium on International Parallel Processing Symposium*, pages 573–579, Washington, DC, USA, 1998. 36

[80] Laxmikant Kale. Charm++. In D. Padua, editor, *Encyclopedia of Parallel Computing.* Springer Verlag, 2011. 180

[81] G. Kestor, L. Dalessandro, A. Cristal, 4 Scott, M. L., and O. S. Unsal. Interchangeable back ends for stm compilers. In *The 6th ACM SIGPLAN Workshop on Transactional Computing*, 2011. 6

[82] G. Kestor, R. Gioiosa, T. Harris, A. Crystal, O. Unsal, I. Hur, and M. Valero. STM2: A parallel STM for high performance simultaneous multithreading systems. In *Proceedings of the 20th IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 221–231, 2011. 6, 92, 95, 119

[83] G. Kestor, R. Gioiosa, O. S. Unsal, A. Cristal, and M. Valero. Enhancing the performance of assisted execution runtime systems through hardware/software techniques. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 153–162, 2012. 7

[84] G. Kestor, R. Gioiosa, O. S. Unsal, A. Cristal, and M. Valero. Hardware/software techniques for assisted execution runtime systems. In *The 2nd Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*, 2012. 7

[85] G. Kestor, O. S. Harris, T. Unsal, A. Cristal, and S. Tasiran. T-Rex: A dynamic race detection tool for c/c++ transactional memory applications. In *under submission the 23th IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2013. 7

[86] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I Hur, and M. Valero. RMS-TM: A comprehensive benchmark suite for transactional memory systems. In *Proceeding of the International Conference on Performance Engineering*, pages 335–346, 2011. 6, 50, 55, 72, 73

[87] G. Kestor, S. Stipic, O. S. Unsal, A. Cristal, and M. Valero. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *The 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009. 6

[88] G. Kestor, S. Tasiran, O. S. Unsal, and A. Cristal. TRADE: Precise dynamic race detection for scalable transactional memory systems. In *under submission the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, 2013. 7

[89] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. A. Yelick. ExaScale computing study: Technology challenges in achieving exascale systems. Technical Report DARPA-2008-13, DARPA IPTO, September 2008. 73

[90] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005. 3

[91] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 209–220, 2006. 4, 21

[92] Christoph Lameter. Effective synchronization on Linux/NUMA systems. In *Proceedings of the Gelato Federation Meeting*, San Jose, CA, USA, May 2005. 22, 81

[93] L. Lamport. Time clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, 1978. 152

[94] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symp. on Code Generation and Optimization*, Palo Alto, CA, USA, 2004. 65

[95] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *The 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009. 5, 50, 86

[96] B. Lewis and D. J. Berg. *Multithreaded programming with Pthreads*. Prentice-Hall, Inc., 1998. 4

[97] B. Liang and P. Dubey. Recognition, mining and synthesis moves computers to the era of tera. In *Technology@Intel Magazine*, pages 1–10, 2005. 28, 29

[98] S. S.W. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 117–128, 2002. 70, 95

[99] Y. Liu, W. Liao, and A. Choudhary. A fast high utility itemsets mining algorithm. In *Proceedings of the 1st International Workshop on Utility-based Data Mining*, pages 90–99, Chicago, IL, USA, 2005. 36

[100] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005. 143, 166

[101] K. Luo, M. Franklin, S. S. Mukherjee, and A. Seznec. Boosting SMT performance by speculation control. In *Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium*, pages 2–, 2001. 93

[102] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. MLP-Aware Dynamic Cache Partitioning. *International Conference on High Performance Embedded Architectures and Compilers*, 2008. 93

[103] P. D. V. Mann and U. Mittaly. Handling OS jitter on multicore multithreaded systems. In *Proceedings of the 2009 IEEE Inter. Symp. on Parallel and Distributed Processing*, pages 1–12, 2009. 96, 119

[104] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *The 1st ACM SIGPLAN Workshop on Transactional Computing*. 2006. 19, 27

[105] V. J. Marathe, M. F. Spear Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *Proceedings of the International Conference on Parallel Processing*, Portland, OR, USA, Sept. 2008. 22

[106] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5, 2006. 12, 128

[107] H. M. Mathis, A. E. Mericas, J. D. McCalpin, R. J. Eickemeyer, and S. R. Kunkel. Characterization of simultaneous multithreading (SMT) efficiency in POWER5. *IBM J. Res. Dev.*, 49(4/5):555–564, 2005. 3, 109, 119

[108] M. Mehrara, J. Hao, P. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 166–176, 2009. 73, 87, 88

[109] M. Mehrara and S. A. Mahlke. Dynamically accelerating client-side web applications through decoupled execution. In *Proceedings of the 9th IEEE International Symp. on Code Generation and Optimization*, pages 74–84, 2011. 70, 95, 121, 183

[110] V. Menon, S. Balensiefer, T. Shpeisman, A. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for java stm. In *Proceedings of the annual symposium on Parallelism in algorithms and architectures*, pages 314–325, 2008. 23, 126, 127, 130, 131, 132, 134, 155, 158, 160

[111] M. R. Meswani and P. J. Teller. Evaluating the performance impact of hardware thread priorities in simultaneous multithreaded processors using SPEC CPU2000. In *Workshop on Operating System Interference in High Performance Applications*, 2006. 96, 119

[112] Sun Microsystem. OpenSPARCTM T2 Core Microarchitecture specification. 70

[113] M. Milovanovi'c, R. Ferrer, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguad'e, and M. Valero. Multithreaded software transactional memory and openmp. In *Proceedings of the Workshop on Memory performance: Dealing with Applications, systems and architecture*. 87, 88

[114] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 69–80, 2007. 4, 21

[115] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, Austin, TX, USA, 2006. 17, 20, 27, 41

[116] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, San Jose, CA, USA, 2006. 15, 28

[117] M. M"uller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 154–159, San Diego, CA, USA, 2003. 36

[118] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, 2006. 152

[119] R. Narayanan, B. "Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *Proceedings of the International Symposium on Workload Characterization*, pages 182–188, San Jose, CA, USA, 2006. 28

[120] Y. Ni, V. S. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. Eliot B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2007. 15, 28

[121] Y. Ni, A. Welc, A. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian.

Design and implementation of transactional constructs for C/C++. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications*, Oct. 2008. 50, 63

[122] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apr'a. Advances, applications and performance of the global arrays shared memory programming toolkit. *International J. High Perform. Comput. Appl.*, 20:203–231, May 2006. 180

[123] M. Olszewski, J. Cutler, and J. G. Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2007. 23

[124] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11, 1996. 3

[125] OpenMP Architecture Review Board. The OpenMP specification for parallel programming. Available at http://www.openmp.org. 179

[126] OProfile - a system profiler for linux. Available at http://oprofile. sourceforge.net/. 39

[127] Oracle database system. http://www.oracle.com/technology/products/timesten/index.html. 12

[128] C. Perfumo, N. Sonmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. In *Proceedings of the 5th Conference on Computing Frontiers*, pages 67–78, Ischia, Italy, 2008. 39, 46

[129] C. Perfumo, O. Unsal, A. Cristal, and M. Valero. TxFS: Transactional file system. Technical Report UPC-DAC-RR-CAP-2010-12, Department of Computer Architecture, Universitat Politecnica de Catalunya, 2010. 34

[130] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualize and analyze parallel code. Technical report, In WoTUG-18, 1995. 100

[131] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the International Symposium on Computer Architecture*, 2005. 20

[132] James Reinders. Transactional synchronization in Haswell. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell. 180

[133] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the Annual Symposium on Parallel Algorithms and Architectures*, pages 221–228, 2007. 4, 5, 19, 22, 27, 65, 77, 81, 119, 125, 126, 131, 132

[134] RMS-TM - BSC Microsoft Benchmark Suite for TM systems. Available at http://www.bscmsrc.eu/software/rms-tm. 48

[135] B. Saha, A.-R. Adl-tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, pages 187–197, 2006. 19, 27, 29, 33

[136] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 161–172, 2007. 125

[137] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, 1997. 152, 157, 162

[138] M. Schindewolf, A. Cohen, W. Karl, A. Marongiu, and L. Benini. Towards transactional memory support for GCC. In *GCC Research Opportunities Workshop*, 2009. 19, 64, 132, 180

[139] K Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. Technical report, University of Minnesota, Minneapolis, 1999. 99

[140] F. T. Schneider, V. Menon, T. Shpeisman, and A. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 181–194, 2008. 12, 129

[141] D. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 285–297, 1989. 152

[142] T. Shpeisman, A. Adl-Tabatabai, R. Geva, Y. Ni, and A. Welc. Towards transactional memory semantics for C++. In *Proceedings of the 21st Ann. Symp. on Parallelism in algorithms and architectures*, 2009. 136

[143] T. Shpeisman, V. Menon, A. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in stm. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 78–88, 2007. 12, 129

[144] A. Shriraman, S. Dwarkadas, and M. L. Scott. Implementation tradeoffs in the design of flexible transactional memory support. *J. Parallel Distrib. Comput.*, Oct. 2010. 21

[145] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 multicore server processor. *IBM J. Res. Dev.*, pages 191–219, May 2011. 92, 93, 94

[146] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, (4/5):505–521, 2005. 93

[147] Rochester software transactional memory runtime. www.cs.rochester.edu/research/ synchronization/rstm/. 50, 131

[148] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the Annual Symposium on Principles of Distributed Computing*, pages 338–339, 2007. 76

[149] M. F. Spear, A. Shriraman, L. Dalessandro, and M. L. Scott. Transactional mutex locks. In *The 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009. 81, 119

[150] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and exploiting inevitability in software transactional memory. In *Proceedings of the 37th International Conference on Parallel Processing*, pages 59–66, Washington, DC, USA, 2008. 28

[151] Michael F. Spear. Lightweight, robust adaptivity for software transactional memory. In *Proceedings of the 22nd Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010. 66

[152] Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and exploiting inevitability in software transactional memory. In *Proceedings of the International Conference on Parallel Processing*, Portland, OR, USA, Sept. 2008. 51

[153] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating systems*, pages 257–268, 2000. 87

[154] B. C. Teixeira, J. Lourenco, and D. Sousa. A static approach for detecting concurrency anomalies in transactional memory. In *Proceedings of InForum 2010*, 2010. 152

[155] S. Tomi'c, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 145–155, 2009. 20, 27

[156] S. Tomi'c, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 145–155, 2009. 29, 33, 34, 40

[157] UPC Consortium. UPC specifications. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005. 180

[158] C. Von Praun and T. R. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 70–82, 2001. 152

[159] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188, 1991. 3

[160] C. Walshaw and M. Cross. Dynamic mesh partitioning and load-balancing for parallel computational mechanics codes. *Computational Mechanics Using High Performance Computing, Saxe-Coburg Publications, Stirling*, 2002. 99

[161] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 34–48, Washington, DC, USA, 2007. 64, 180

[162] M. Ware, K. Rajamani, M. Floyd, B. Brock, J.C. Rubio, F. Rawson, and J.B. Carter. Architecting for power management: The IBM POWER7 approach. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2010. 70, 80, 92, 94, 179

[163] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, S. Margherita Ligure, Italy, 1995. 27, 28, 46

[164] X. Xie and J. Xue. Acculock: Accurate and efficient detection of data races. *Code Generation and Optimization, IEEE/ACM International Symposium on*, pages 201–212, 2011. 157

[165] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 43, 1996. 35

[166] C. B. Zilles, J. S. Emer, and G. S. Sohi. The use of multithreading for exception handling. In *Proceedings of the 32nd annual ACM/IEEE International Symp. on Microarchitecture*, 1999. 70, 87, 95, 121, 183

[167] F. Zyulkyarov, A. Cristal, S. Cvijic, E. Ayguade, M. Valero, O. Unsal, and T. Harris. Wormbench: A configurable workload for evaluating transactional memory systems. In *Proceedings of the 9th Workshop on Memory Performance*, pages 61–68, Toronto, ON, Canada, 2008. 46

[168] F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguad'e, T. Harris, and M. Valero. Atomic quake: Using transactional memory in an interactive multiplayer game server. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–34, Raleigh, NC, USA, 2009. 27, 28, 46

[169] F. Zyulkyarov, T. Harris, O. S. Unsal, A. Cristal, and M. Valero. Debugging programs that use atomic blocks and transactional memory. In *Proceedings of the Symposium on Principles and practice of parallel computing*, 2010. 125