UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Algorithms for Enhancing the Performance Robustness of Transactional Memory Systems

Nuno Miguel Lourenço Diegues

Supervisor: Doctor Paolo Romano

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering

Jury final classification: Pass with Distinction and Honour

Jury

Chairperson: Chairman of the IST Scientific Board

Members of the Commitee:
Doctor Leonel Augusto Pires Seabra de Sousa
Doctor Paolo Romano
Doctor João Manuel dos Santos Lourenço
Doctor Aleksandar Dragojevic

2016

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Algorithms for Enhancing the Performance Robustness of Transactional Memory Systems

Nuno Miguel Lourenço Diegues

Supervisor: Doctor Paolo Romano

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering

Jury final classification: Pass with Distinction and Honour

Jury

Chairperson: Chairman of the IST Scientific Board

Members of the Commitee:

Doctor Leonel Augusto Pires Seabra de Sousa, Professor Catedrático,
Instituto Superior Técnico, Universidade de Lisboa
Doctor Paolo Romano, Professor Associado, Instituto Superior Técnico, Universidade de Lisboa
Doctor João Manuel dos Santos Lourenço, Professor Auxiliar,
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
Doctor Aleksandar Dragojevic, Researcher, Microsoft Research, United Kingdom

Funding Institutions
Fundação para a Ciência e Tecnologia
European Commission
COST (European Cooperation in Science and Technology)

2016

Acknowledgments

First and foremost, I wish to deeply thank my advisor, Professor Paolo Romano, without whom this work would never have been materialized. One of the most important days — on the course to delivering this dissertation — was actually among the very early days of my doctoral studies, when I struggled to obtain a grant, until I knocked on his office door and he accepted to supervise my studies and to provide me with a grant. In hindsight, I recall being close to deciding to skip further studies, so I am eternally grateful for him to have provided me with this opportunity, at a time when he barely knew me at all. Several years followed of intense work, where he guided me through the uncertainties of research, while always being available to push for yet another idea. I still envy his relentlessness to overcoming any problem, which was crucial for me to have come this far. I am also deeply thankful for his support to allow me to attend many conferences and workshops, which has helped shape my academic persona. Finally, his openness for pursuing internships in the course of the studies, at the loss of some months of productivity, is also noteworthy and noble. I am truly honored to having been his student, and I can only hope to having matched his high standards.

I would also like to thank Professor Luís Rodrigues, whose advice has influenced deeply my decisions over the years. He was a constant voice of reason that helped to complement my advisor's guidance. It was always encouraging when he entered our room late in the evening, when almost everyone was gone for the day, and still cheered us up to find our way through the challenges we were facing. All of this with his usual and contagious laughing.

During this long journey I shared many adventures with several great colleagues. Pedro Ruivo whose help in my first works was undoubtedly essential, as I dove into the code base of Infinispan, pretty much clueless about what I had to do at all. João Paiva for having been a great team mate in our fruitful collaboration with Muhammet. Nuno Machado, Xavier Vilaça and Maria Couceiro, for all those days we spent together in the same room; I shall miss room 612 until the end of my life. And finally for Hugo Rodrigues and Pedro Mota, with whom I have paved the way through the PhD grind. I cannot possibly name everyone that has influenced my

PhD journey and with whom I shared important experiences over the years: to all those left, I thank you for the good times.

To my mother Luísa and grandmother Laurinda, I am deeply grateful for all your support over the years. I still recall when you enticed me to pursuing the PhD, even when faced with its initial challenges and disappointments. I feel really proud to have reached this day while sharing almost every day with you. This accomplishment is as much mine as it is yours.

Finally, to Beatriz Ferreira, for having stood by me in the course of this journey. I am so happy that we got to share this life-shaping experience together, and I can only begin to express my gratitude for your complacent acceptance in every time I was working late to pushing for another paper.

Lisbon, September 2016

Nuno Diegues

The work presented in this thesis was supported by the following funding agencies:

- INESC-ID through the multi-annual funding of the PIDDAC Program funding grant under projects PEst-OE/EEI/LA0021/2011 and PEst-OE/EEI/LA0021/2013.
- European Commission through the European STReP project Cloud-TM (FP7/257784) and the Euro-TM COST Action (IC1001).
- Fundação para a Ciência e Tecnologia (FCT) via the individual Doctoral grant SFRH/BD/94289/2013 and projects Green-TM (EXPL/EEI-ESS/0361/2013), SpecSTM (PTDC/EIAEIA/122785/2010) and Aristos (PTDC/EIA-EIA/102496/2008).

Thesis Publications List

Most of the material presented in this work can also be found in the following publications (here in chronological order):

Journals

1. N. Diegues and P. Romano. Time-Warp: Efficient Abort Reduction in Transactional Memory. In ACM Transactions on Parallel Computing (TOPC), Vol. 2, Issue 2, Article 12, July 2015.

2. N. Diegues and P. Romano. Bumper: Sheltering Distributed Transactions from Conflicts. In Elsevier Future Generation Computer Systems (FGCS), Vol. 51, October 2015.

3. N. Diegues and P. Romano. Self-tuning Intel Restricted Transactional Memory. In Elsevier Parallel Computing (ParCo), Vol. 50, December 2015.

Conferences

1. N. Diegues and P. Romano. Bumper: Sheltering Transactions from Conflicts. In Proceedings of the 32^{nd} IEEE International Symposium on Reliable Distributed Systems (SRDS), 2013.

2. N. Diegues and P. Romano. Time-Warp: Lightweight Abort Minimization in Transactional Memory. In Proceedings of the 19^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2014.

3. N. Diegues, P. Romano and Luís Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. In Proceedings of the 23^{rd} International Conference on Parallel Architectures and Compilation Techniques (PACT), 2014 (**Runner-up for the Best Paper Award**).

4. N. Diegues and P. Romano. Self-Tuning Intel Transactional Synchronization Extensions. In proceedings of the 11^{th} USENIX International Conference on Autonomic Computing (ICAC), 2014 (**Best Paper Award**).

5. N. Diegues, P. Romano and S. Garbatov. Seer: Probabilistic Scheduling for Hardware Transactional Memory. In Proceedings of the 27^{th} ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2015.

6. D. Didona, N. Diegues, R. Guerraoui, A. Kermarrec, R. Neves and P. Romano. ProteusTM: Abstraction Meets Performance in Transactional Memory. In Proceedings of the 21^{st} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2016.

Resumo

O aparecimento de processadores com múltiplos núcleos levou a uma mudança significativa de paradigma de desenvolvimento de software. Como tal, uma aplicação de software que seja desenvolvida hoje só executará mais rapidamente nos processadores futuramente lançados, caso seja devidamente paralelizada por via a tirar partido do crescente número de núcleos em cada processador. O desenvolvimento de tais aplicações, ditas concorrentes, tem vindo a representar um desafio considerável, nomeadamente pelas várias complexidades associadas à sincronização correcta de acessos concorrentes aos dados mutáveis partilhados da aplicação.

Ao longo da última década, a Memória Transacional (MT) tem vindo a ser gradualmente considerada como um dos paradigmas de programação mais promissores para facilitar o desenvolvimento de aplicações concorrentes. A MT disponibiliza aos programadores uma interface simples através da qual eles declaram porções do código da aplicação que devem executar de forma atómica. A implementação da MT executa cada uma destas porções de código encapsulada numa transação, como tal, providenciando uma ilusão de que a aplicação executa todo código apenas sequencialmente, apesar de as transações poderem (e deverem) ser efectivamente executadas de forma concorrente.

Durante estes últimos anos pudemos testemunhar avanços tremendos na área de MT, nomeadamente com a sua consagração em linguagens de programação amplamente utilizadas, assim como na sua disponibilização em implementações de hardware em vários processadores de múltiplos núcleos em milhões de computadores. De uma forma geral, a comunidade académica tem vindo a aclamar a simplicidade da programação de aplicações concorrentes com recurso a MT. No entanto, a outra promessa de alto desempenho e escalabilidade da MT, tem sido frequentemente questionada como fracassada. De facto, as implementações de MT têm ainda algumas falhas. Ao procurar obter o máximo de simplicidade possível, os sistemas de MT têm vindo a sacrificar essa promessa, resultando tipicamente em sistemas cuja materialização desaponta em termos de desempenho e escalabilidade face ao potencial alto que lhe é reconhecido. Exemplos destes problemas vão desde o excessivo número de recomeços de transações até à

rigidez dos algorithmos que tendem a obter bom desempenho nalgumas aplicações e desempenho sub-óptimo noutras.

O objectivo principal desta dissertação é o de criar algoritmos inovadores que melhoram o desempenho de sistemas de MT de forma robusta, i.e., com ganhos em diferentes cargas de trabalho heterogéneas e sem a intervenção manual por parte do programador para a escolha de parâmetros e configurações. A capacidade de obter desempenho robusto a mudanças é particularmente relevante para resolver um dos maiores impedimentos na adoção de MT, como mencionado acima. A garantia de total automatização e transparência para o programador é também da maior importância, para que a facilidade de programação do paradigma de MT seja preservada. Este aspeto distingue notoriamente este trabalho do estado da arte que tenta melhorar o desempenho de sistemas de MT.

Abstract

The advent of multi-core processors has driven a major paradigm shift in the development of software. An application that is developed today will only run faster with future processors if it is parallelized to take advantage of the ever-growing number of cores of modern processors. Developing concurrent applications, however, is a hard problem, especially due to the complexity associated with correctly and effectively synchronizing concurrent accesses to the shared mutable data.

Over the past decade, Transactional Memory (TM) has emerged as one of the most promising programming paradigms to ease the development of concurrent applications. TM provides programmers with a simple interface, which simply requires them to declare the portions of their code that should be executed atomically. The TM runtime executes atomic code blocks within transactions, which provide the illusion that the application executes sequentially, even though its transactions are actually processed concurrently.

The past few years have resulted in tremendous advances in the area of TM, with its standardization in mainstream languages, as well as hardware implementations in several commodity multi-core processors. To a large extent, the research community has acknowledged the simplicity of programming concurrent applications with TM. However, its performance and scalability have been often disputed. Indeed, existing TM implementations are far from perfect, and they often fall prey of several inefficiencies, which can ultimately lead TM to deliver unsatisfactory performance and scalability levels. These issues range from unnecessary transaction aborts, to the strictness of the algorithms that perform well in some workloads and sub-optimal in others.

The main goal of this dissertation is to create novel algorithms that improve the performance of TM systems in a robust way, i.e., with gains across heterogeneous types of workloads and without manual intervention from the programmer to tune parameters or choose configurations. The ability to deliver robust performance is particularly relevant to address one of the main concerns with the adoption of TM, as identified above. The use of fully automatized self-tuning

approaches is also of the utmost importance, so that the ease of programming of the TM paradigm is preserved. This is an aspect that notably distinguishes this work from prior state of the art also aimed to improve the performance of TM.

Palavras Chave

Memória Transacional, Desempenho, Ajustamento Automático, Gestão de Concorrência, Recomeço de Transacções, Memória Transacional por Software, Memória Transacional por Hardware

Keywords

Transactional Memory, Performance, Self-tuning, Concurrency Control, Transaction Restarts, Software Transactional Memory, Hardware Transactional Memory

Index

List of Figures

List of Tables

1 Introduction

1.1 Context

In 2004 the president of Intel announced a dramatic shift in the strategy of the company: "We are dedicating all of our future product development to multi-core designs". Over ten years later, multi-core processors have become the reference architecture for modern computing systems, with an uncontested ubiquity across many types of devices.

To fulfill the potential of such multi-core processors, programmers are now obliged to write concurrent applications, as otherwise their applications will not scale with the new generations of processors — which typically have more cores, but each one executes at the same speed of past generations. As a consequence of this paradigm shift, today, the challenge of synchronizing concurrent accesses to shared data held in-memory is no longer faced only by the specialists of parallel programming, but also by the mass of common programmers, who wish to exploit in a simple way the full potential of modern processor architectures.

The idea of Transactional Memory (TM) [Herlihy and Moss, 1993] is to provide an answer to this challenge, namely to allow programmers to easily obtain correct programs, in which many concurrent threads manipulate shared data, without sacrificing scalability or performance. To do so, TM systems typically provide an illusion of a sequential application to the programmer, even though she may exploit concurrency by issuing concurrent transactions encompassing atomic blocks of code. Furthermore, TM systems often rely on optimistic concurrency control mechanisms that allow to unveil much of the parallelism that gets neglected by conservative approaches such as those based on coarse-grained locking.

Contrarily to the conventional lock-based synchronization methodologies, in which programmers identify shared data and specify how to synchronize concurrent accesses to it, the TM paradigm requires only to identify which portions of the code have to execute atomically, and not *how* atomicity should be achieved.

Although the initial proposal of TM [Herlihy and Moss, 1993] was somewhat overlooked by the research community, the advent of multi-core processors generated additional motivation to explore simple concurrent programming paradigms to manage in-memory shared data, which brought TM to the forefront [Herlihy et al., 2003b].

Many alternative TM systems have been proposed since then. One fundamental difference among them is that some are pure Software TMs (STMs), which are implemented as user-level libraries [Shavit and Touitou, 1995]. Others, referred to as Hardware TMs (HTMs), rely entirely on the logic embedded in the hardware of the processor [Yoo et al., 2013]. Finally, others rely on both software and hardware mechanisms, and are, therefore, referred to as Hybrid TMs (HyTMs) [Calciu et al., 2014a].

The initial implementation proposed for TM was hardware-based (i.e., an HTM) and it relied on an extension of the processor's cache coherency protocol to support hardware transactions. However, given the unavailability of commercial processors with HTM support, most of the research in this area has initially focused on the development of STMs, as flexible prototypes that allowed the exploration of many designs and implementations.

More recently, the relevance of TM in the industry has been significantly amplified by the achievement of two main milestones. On the one hand, the standardization of programming constructs for TM in popular programming languages (such as C/C++ [Ni et al., 2008]). On the other hand, the launch of mainstream processors with HTM (namely, by Intel [Yoo et al., 2013] and IBM [Adir et al., 2014, Cain et al., 2013, Nakaike et al., 2015, Wang et al., 2012a]). These two technologies, embodying TM in compilers and processors, are now available to millions of programmers across the globe, turning TM into a leading paradigm to simplify concurrent programming.

1.2 Thesis Statement

TM systems have evolved superbly over the past decade, greatly compelled by the motivation to take advantage of the multi-core processors that become mainstream during this time. At the same time, these TM systems are gaining increasing adoption, with their recent commercialization and standardization.

Recall that the premise of TM is two-fold: 1) to enable easy development of concurrent

applications, and 2) to unfold the potential to exploit as much parallelism as possible in the workload and improve performance.

The former is possible thanks to the adoption of the familiar abstraction of transactions to delimit portions of code that should execute atomically, i.e., appear to the programmer as a single, instantaneously executable instruction — this allows the programmer to reason about the application as if it executed sequentially, even though the atomic blocks are executed concurrently. The benefits of ease of programming stemming from the adoption of the TM paradigm, when compared to conventional lock-based approaches, have garnered an undisputed consensus [Lupei et al., 2010, Pankratius and Adl-Tabatabai, 2011, Rossbach et al., 2010]. One tangible effect of this consensus was the aforementioned standardization of TM in popular programming languages and its adoption in commercial processors.

The second premise of TM, i.e., enhancing performance, is enabled by speculatively executing transactions in parallel and relying on optimistic concurrency control techniques that transparently abort and restart the conflicting transactions.

Indeed, existing implementations of TM are far from perfect, and their actual performance and efficiency has been questioned and criticized in the past [Cascaval et al., 2008]. While striving for ease of usage, namely by running concurrent transactions in a way that provides an illusion of a sequential execution, TM systems can fall prey of several inefficiencies. As a result, it is often the case that a given TM delivers quite contrasting performance and scalability across varying workloads, yielding high performance only in some restricted workloads. This lack of robust performance is the key weakness factor that is still identified to existing TM systems — ideally, the performance of TM would be robust, in the sense that it would perform consistently across workloads and applications, without the need for manual tuning of parameters or careful choice of the algorithms used. In practice, as it shall be demonstrated in this dissertation, a given TM system tends to deliver high-performance for some cases and to perform sub-optimally, when compared with alternative TMs.

Unfortunately, this issue has not been solved by the recently available commodity HTMs: as we shall see in this work, it is often the case that STMs outperform HTMs, which defies the assumption that hardware-based implementations should be faster than their counter-parts in software. This is, in part, a result of the intense research in the area of STM, before the arrival of commodity HTMs, which generated several robust software-based implementations over the

years. However, it is also in part due to the unfortunate reality that current HTMs are faced with significant restrictions, and thus their performance is also restricted in some cases.

This dissertation seeks to demonstrate the following thesis. It is possible to create TM systems with **robust performance** across workloads and applications while **preserving the simplicity and ease of use** of the TM abstraction.

Indeed, new paradigms — as is the case for the TM paradigm — often have to provide a significant advantage in order to become widely adopted, as incremental changes are not appreciated enough to justify the change. With this thesis, the intent is to pave the way to providing one of the missing key factors in TM to turn it into a mainstream paradigm for concurrent programming, i.e., performance that is robust to varying workloads and types of applications and that scales with the available number of cores. An extremely significant aspect of the thesis is to do so in ways that are fully automatic and transparent to the programmer. This is an aspect of the utmost importance to preserve that significant feature of the TM paradigm, which notably distinguishes this work from prior state of the art trying to improve the performance of TM.

1.3 Outline of the Contributions

In order to pursue the statement defined above, this dissertation follows two main approaches. On one hand, new TM algorithms are proposed and implemented, whose results demonstrate that it is possible to achieve robust performance in workloads that challenge previous state of the art solutions. On the other hand, several self-tuning mechanisms are also proposed to enrich the TM run-time and adapt several key mechanisms used in TM implementations, so that they better fit the characteristics of the underlying workload being executed.

In the following we summarize the key contributions of this dissertation that achieve the goal identified above:

• *A comparison of existing TMs* (Chapter 3): The advent of commodity HTMs, whose release by Intel and IBM was contemporary with the work conducted in this dissertation, raised a compelling research question to quantitatively assess the efficiency of HTM implementations with respect to alternative synchronization mechanisms, including software and hybrid TM implementations as well as conventional lock-based mechanisms. Indeed, it was of the utmost

importance to establish a fair comparison for the state of the art of TM implementations, when framed under the availability of these new HTMs. We tackled this research question by carrying out the largest experimental study on TM-based synchronization to date. As part of the results, we showed for the first time that the commodity HTM released by Intel was not the providing the robust performance that one would expect, as for more than half the workloads we tested it performed similarly or worse than the best available STMs. At the same time, for workloads with small transactions and low contention, HTM delivered unparalleled performance given that it has negligible software instrumentation on the application code. Finally, the supposedly "best of both worlds" idea behind HyTMs was also shown to fall significantly short, as combining HTM and STMs safely to allow hardware and software transactions to run simultaneously ends up polluting the fast path (of hardware transactions) with software instrumentation that generates overhead and spurious aborts. These experimental findings motivated the pursue of performance improvements on both fronts: both to improve existing — and well studied — STM algorithms, as well as to propose new ideas to maximize the performance achievable with the new HTM support.

• TIME-WARP — *Reduction of aborts in STMs* (Chapter 4): In TM systems, transactions are typically executed in parallel and regulated by an optimistic concurrency control mechanism that rejects inconsistent executions by aborting some transactions. In the scope of TM, the reference consistency criteria are strictly stronger than that of Serializability [Guerraoui and Kapalka, 2008, Imbs and Raynal, 2012], which, roughly speaking, ensure that the effects of concurrency are not exposed to applications, which observe solely snapshots of shared data that are equivalent to those of a serial execution. To also strive for high-performance, it is desirable to accept all consistent executions, and thus avoiding spurious aborts. Classic literature has evidenced the algorithmic complexity (and practical impossibility) of doing so [Papadimitriou, 1979]. Hence the challenge is on how to reduce spurious aborts without incurring overheads so high that they outweigh the gains stemming from the reduction of aborts. We tackle this challenge by devising an STM algorithm that drastically reduces aborts in high contention scenarios while remaining competitive with the most efficient state of the art STMs in low contention workloads. All of this without imposing any additional burden on the programmer — i.e., it is fully automatic and still ensures strong consistency as desirable and described above.

• TUNER — *Improving the performance of HTM-based systems via self-tuning* (Chapter 5):

As established by the first contribution of this dissertation, HTM is not a panacea for the challenges inherent to concurrent programming, mainly because those solutions are limited by hardware constraints. As a result of these constraints, existing HTM implementations must have a software fall-back path that correctly implements the TM abstraction when the HTM cannot do so. Furthermore, both paths, namely the hardware and software ones, must execute correctly in concurrency. In this work we shed light on a relevant issue with great impact on the performance of such HTMs: the correct tuning of the logic that regulates how to cope with failed hardware transactions. We show that the optimal tuning of this policy is strongly workload dependent, and that the relative difference in performance among the various possible configurations can be remarkable. We address this issue by introducing a simple and effective approach that aims to identify the optimal fall-back configuration at runtime via lightweight reinforcement learning techniques. The proposed technique requires no off-line sampling of the application, and can be applied to optimize HTMs and HyTMs.

• SEER — *Reduction of aborts in HTMs via scheduling* (Chapter 6): Due to the optimistic nature of TM implementations, transactions may have to be aborted and restarted, so that correctness is ensured. This phenomenon is exacerbated by the limitations of HTM, which generate more aborts when the working-set is too large to fit the hardware limits, and which generate also more aborts due to conflicts than STMs due to the coarser conflict detection granularity. As a result, it is up to the software library that regulates the HTM usage to ensure progress and optimize performance. A well known technique in the literature to do so is that of scheduling transactions: the scheduler allows transactions to start when it is confident that the concurrently running transactions will not conflict with it. STM solutions can provide precise information to feed the decisions of the scheduler, due to their instrumentation in the application that collects data about the read and write accesses of each transaction. HTM solutions, however, provide only coarse feedback regarding the type of abort (e.g., a data conflict abort versus a forbidden instruction abort) and cannot therefore employ state of the art schedulers that were designed with STM in mind. To address this crucial issue for HTMs, and consequently reduce the aborts that they suffer, this dissertation proposes a software scheduler called SEER. SEER leverages an on-line probabilistic inference technique that identifies the most likely conflicting pairs of transactions, and establishes a dynamic (i.e., varying over time) locking scheme to serialize transactions in a fine-grained manner.

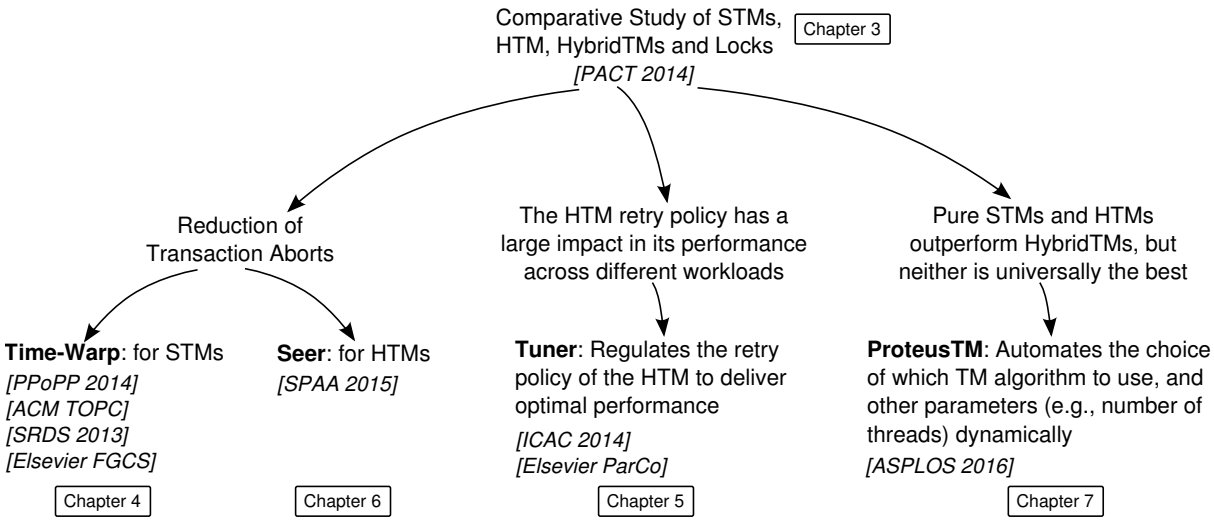


Figure 1.1: Summary of the contributions of this dissertation to the thesis of providing robust high-performance for TM while preserving both its simple interface and correctness criteria. Attached to each contribution, we list also the corresponding publications (by conference or journal acronym and year), as well as the chapter where the respective work is presented.

• ProteusTM — An Adaptive High-Performance TM (Chapter 7): With the initial study presented in this dissertation, we show that HTM and STM perform optimally in different workloads, with HyTM falling short of being the desirable way to reconcile the former two. To tackle such cases, where the workload may change over time, we propose to adaptively choose between HTM and STM algorithms. Furthermore, we show that the best possible performance is also dependent on other configurations of the TM system — besides the algorithm itself, also the number of active threads, and internal parameters such as the contention management are also highly important — making this a multi-dimensional optimization problem. Imposing this burden on developers is unacceptable, both for its intractability in terms of number of possible configurations to explore manually, but also because no static solution can cope with workloads varying over time. With ProteusTM, we address the challenge of automatically identifying the best TM implementation and configuration for a given workload. Our proposed system hides, behind the simple TM interface of atomic blocks, a large library of dynamically tunable (and highly extensible) TM implementations. Under the hood, ProteusTM leverages an innovative, multi-dimensional online optimization scheme, combining two popular machine learning techniques: Collaborative Filtering and Bayesian Optimization. These black-box learning techniques are used to dynamically drive, in a fully transparent way, the choice of underlying TM implementation and the tuning of several configuration parameters. ProteusTM has been integrated in a leading open source compiler for C/C++ (GCC), and extensively evaluated using a suite

of 11 applications and 4 micro-benchmarks. Our study shows that ProteusTM obtained average performance that is $< 3\%$ from optimal, and gains up to $100\times$ over static alternatives.

This set of challenges that we faced during the work for the dissertation, and corresponding contributions, are summarized in Figure 1.1.

Before delving into each of these contributions, we first present, in Chapter 2, a contextual background on the TM abstraction and its underlying implementations. Chapters 3-7 present our contributions to improve the performance of state of the art TMs — in each of them, we first present a statement of the problem, overview of the solution, and a contextual discussion of the specific related work, before explaining the details of our proposal. Finally, Chapter 8 summarizes this work and identifies possible avenues for future research.

2 Background on Transactional Memory

The past decade has brought remarkable advances in the area of TM, including its progressive adoption by the industry [Karnagel et al., 2014, Intel Corporation, 2009]. This has resulted in a wide range of works, encompassing very different implementations of TM, which we review in this chapter. First, we introduce the TM abstraction in Section 2.1, as an idea with a wide applicability to help address the challenges posed by concurrent programming. Next, in Section 2.2, we describe theoretical guarantees that are provided by TMs. Then, we proceed with an overview of the various types of TM implementations in Sections 2.3-2.6. Finally, we present the benchmarks and applications used in the various evaluations conducted in the course of this work, in Section 2.7.

2.1 The Transactional Memory Abstraction

The idea underlying TM was proposed concurrently by [Stone et al., 1993] and [Herlihy and Moss, 1993]. To some extent, both proposals sought to simplify concurrent programming by providing a primitive that allowed multiple atomic reads and writes. Both proposals were also quite system-centric, and proposed implementations that would extend the cache coherence protocol available in processors.

Although the spotlight may have been more on the implementation side, in fact, these two proposals had coined the TM abstraction, i.e., a concurrent programming paradigm. This paradigm requires programmers to identify only which code portions need to be executed atomically, and not how atomicity is to be enforced. Such atomic portions of code execute as transactions, whose implementation may vary. They contrast with lower level primitives, such as locks, that require the programmer to define how the actual synchronization will take place for accessing each shared datum.

Consider for instance the sample of code shown in Listing 2.1. There we show some simple

Listing 2.1: Example of C++ sample of code using the Transactional Memory abstraction to synchronize accesses to shared memory.

```
1  bool transfer(const Account* source, const Account* dest, int amount) {
2    __transaction_atomic {
3      if (source->balance < amount) {
4        return false;
5      }
6      source->balance -= amount;
7      dest->balance += amount;
8      return true;
9    }
10 }
```

logic in C++ to transfer an amount of money between two ACCOUNT objects atomically. The atomic block, delimiting the code to run in a transaction, provides an effective way to identify that. Contrasting this with a lock-based implementation, one would have to devise a mapping of locks to accounts, and ensure that they were acquired in a consistent order to avoid deadlocks [Herlihy and Shavit, 2008]. Naturally, the programmer could use a single lock to protect all accounts, but that would result in a sequential bottleneck for concurrent transfers between independent pairs of accounts.

It is also worth highlighting that it is no coincidence that we use this specific "__transaction_atomic" syntax for the atomic block: this matches exactly the only thing that a programmer would have to write with the standard C++ programming constructs for TM [Adl-Tabatabai et al., 2012]. Indeed, by raising the level of abstraction, TM has been shown to drastically simplify the development of parallel applications [Pankratius and Adl-Tabatabai, 2011, Rossbach et al., 2010].

Behind this abstraction, there is a myriad of possible implementations, which we overview over several sections later in this chapter. All these implementations of TM have, in common, the fact that the data being shared by different threads is held in main-memory, which explains the name of the abstraction. A transaction is typically structured in three phases:

1. *Start*: this is used to prepare data-structures and meta-data that is private to the thread that is about to execute the transaction. After a thread invokes this, it is usually said to be running a transaction, and as such reads and writes will be performed transactionally.

2. *Accessing data*: to ensure the correctness of concurrent executions, reads and writes per-

formed transactionally usually have to be redirected for handling by the TM system. Some TMs may be completely transparent to applications, as is the case for HTMs, whereas STMs, in contrast, require applications' memory accesses (i.e., reads and writes) to be intercepted via software wrappers (instrumented into the application) that redirect them to a TM library implemented in software. This usually ensures that read- and write-sets are maintained to track the addresses/objects of the program that were read or written transactionally. Naturally, HTMs perform an analogous process, albeit it is embedded in the micro-code of the processor and thus transparent to the application.

3. *Commit*: this phase attempts to consolidate the speculative writes that were recorded during the transactional accesses, thus making them accessible to other transactions (should the commit be successful). This operation may fail when the TM deems the transaction to threaten the correctness criteria condition, in which case the transactional writes are discarded, and the transaction is restarted.

2.2 Theoretical Guarantees

There are several interesting properties that characterize quite distinctively the various implementations of TMs. These can often be analyzed from a theoretical, i.e., algorithmic point of view, and thus provide important means of establishing comparisons between different TMs. We focus on a few of them, whose understanding is crucial to the nature of the contributions of this dissertation.

2.2.1 Correctness Criteria

The abstraction of transactions stems from the well known field of databases, which have used transactions to reach similar goals to those of TM, although many times in sand-boxed domains (e.g., such as Structured Query Language (SQL) statements) — whereas TM operates on arbitrary programs — and with persistence and durability as crucial properties — whereas in TM the shared data lies in-memory and is not concerned with those properties.

Despite those differences, the correctness criteria of TM are highly inspired by those of databases. More specifically, the widely known idea of Serializability establishes that a concur-

rent execution is safe if it is equivalent to one yielded by a serial execution of the system [Papadimitriou, 1979]. However, the formal definition of Serializability enforces this condition only for committed transactions, without specifying anything for transactions that eventually abort. As we shall see, transactions that do not commit are still important for general purpose programs — contrarily to database systems where the environment is much more constrained — and so researchers in TM have identified stronger criteria to provide safer conditions.

By specifying safety only for committed transactions, Serializability allows other transactions (that eventually get aborted) to perform inconsistent reads — that is, read operations may return values that would not be possible in a sequential execution. As such, this allows threads to follow execution paths in their code that the programmer would not have conceived or expected. For instance, a thread running a transaction could read two shared variables x and y, which would normally respect an invariant that $x < y$, and then execute a loop with those variables as lower and upper bound (respectively). However, if the transaction obtains inconsistent values (for instance $x > y$), then it may remain in an infinite loop or issue a memory access at an illegal memory location causing an unrecoverable exception and the crash of the whole application.

Situations like this one have motivated the introduction of Opacity [Guerraoui and Kapalka, 2008] in which: 1) the definition of Serializability is preserved, 2) additionally, all transactions — including live ones that may still abort — must always access a consistent state of the shared data, i.e., a state that would be possible to obtain by a serial execution of committed transactions, and 3) all committed transactions must respect the real-time order, i.e., if T_1 finishes before T_2 starts, then T_1 must appear to have executed — that is, serialized — before T_2.

There is another interesting correctness criterion called Virtual World Consistency (VWC) [Imbs and Raynal, 2012], which is weaker than Opacity, and stronger than Serializability. Whereas in Opacity all transactions must observe the same consistent state, the one resulting from an equivalent serial order of committed transactions, in VWC this is only required for committed transactions. For a transaction that (eventually) aborts, the requirement is weakened so that it must observe state consistent with its causal past (i.e., roughly transactions whose presence it has witnessed via reads over memory they wrote); however this state and causal past is not necessarily the same as that observed by other (eventually) aborted transactions. This means that an aborted transaction may observe a different set of transactions to have committed than the one that another aborted transaction observes.

This subtle change makes VWC weaker than Opacity and, as such, imposes less constraints on the potential concurrency of transactions. All of this while keeping the spirit that motivated Opacity: that is, to ensure all transactions always observe a consistent state, to avoid hazardous situations such as infinite loops and segmentation faults that would not occur in sequential runs of the program.

2.2.2 The Need to Abort Transactions

Guaranteeing any of the correctness criteria identified above would be trivial in a TM implementation that simply aborts all transactions. This motivates the need to define conditions that capture the scenarios in which a transaction cannot abort (and thus must progress and commit successfully).

As a result, the concept of C-Permissiveness was defined [Guerraoui et al., 2008], where C is a correctness criterion (e.g., Opacity): a TM is C-Permissive if it never aborts a transaction unless there is no alternative to preserve the correctness of the concurrent execution according to C. When a transaction is aborted, even though the resulting concurrent execution (if it was committed) was safe with respect to C, then it is said that this was a spurious abort.

While being extremely appealing and desirable, ensuring C-Permissiveness turns out to be impractical due to its algorithmic complexity [Keidar and Perelman, 2009]. To understand the intuition behind this, one can consider a TM to be an online algorithm, in the sense that, on each operation that it receives (e.g., a start, read, write or commit), it has to decide whether to accept it or to abort the transaction. However, the TM has no knowledge on what future operations it will have to execute, which complicate the reasoning that it has to do for every operation.

This led to an alternative notion named Probabilistic C-Permissiveness [Guerraoui et al., 2008], which requires a TM to ensure C-Permissiveness only with some probability. This means that all C safe concurrent executions are acceptable by the TM, but they are not always accepted with certainty (otherwise the TM would be in fact C-Permissive). This allows implementations to use randomization when deciding the outcome of an operation, and thus achieve a meaningful property — albeit a less interesting one — without incurring prohibitive costs. Another weaker, but still similar property, is that of MV-Permissiveness [Perelman et al., 2010], which requires

read-only transactions to never abort.

The notion of Permissiveness seems to capture the intent of a programmer with respect to the need of aborting transactions in TM. However, by being too demanding, it has not been particularly popular in terms of adoption: to the best of our knowledge the has only been one (theoretical) Opaque-Permissive TM algorithm proposed [Keidar and Perelman, 2009].

As such, the idea of Input Acceptance [Gramoli et al., 2010] was proposed to allow to capture the relative power of TMs to reduce the amount of spurious aborts. The key idea is to identify input patterns, encompassing sequences of input TM operations (such as reads and writes) that, when fed to a TM algorithm, cause the abort of at least one transaction. In that case it is said that the TM does not accept that input pattern. Then, by comparing the patterns accepted (or not) by different TMs, it becomes possible to define a hierarchy between them that captures their relative ability to avoid spurious aborts: this is achieved by identifying inclusion (i.e., subset) relationships also between those patterns. Intuitively, this means that if TM_B accepts only a subset of the patterns accepted by TM_A, then TM_A has a higher input acceptance than TM_B and so generates less spurious aborts.

2.2.3 Progress Guarantees

It is also important to define the guarantees of progress for concurrent threads. Namely, because they contend to access shared data, and as such one would like to ensure that not all threads in the program remain waiting for some condition that will never be verified. These concerns are not specific for TM and, in fact, the following properties that we summarize have originated from the literature of concurrent programming in general [Herlihy and Shavit, 2008].

Although the TM abstraction strives to get away from the concept of locking, it happens that many TM implementations actually use locks to internally manage the concurrency between threads accessing shared data transactionally. As such, any practical TM has to ensure at least Deadlock-freedom, which requires that, when trying to grab any lock, at least some thread eventually manages to do so with success. This definition allows specific threads to never succeed meaning that they can be stuck forever. In fact, many TMs have been proposed with this guarantee only. To overcome this limitation, Starvation-freedom enforces that all threads eventually manage to progress when trying to grab any lock.

Other TMs do not actually use locks internally, and tend to resort to the Compare-And-Swap (CAS) primitive or similar, or even have their implementation encoded in the micro-code of the processor in the case of HTMs. For this non-blocking case, the guarantee of Lock-freedom requires that at least one thread makes progress in the system, i.e., in the case of TM this entails that some transaction finishes (note that it does not imply committing — it may be aborted). Similarly, Wait-freedom requires that every thread eventually obtains progress.

2.3 Software Transactional Memory

Software TMs (STM) stemmed mostly from the need to rapidly prototype new ideas in research. The inherent difficulty of doing so purely in hardware, or the inaccuracy of doing so via hardware simulations, led to the creation of many STMs (the first one being [Shavit and Touitou, 1995], curiously long before the multi-core era). More recently, HTMs turn out to be somewhat limited in terms of applicability, which is another motivation for still relying on STMs.

In an STM, read and write transactional accesses are instrumented to be intercepted by the STM library. Naturally, this can lead to large overheads, for which reason applications are typically instrumented to trace only shared memory accesses. This can be achieved by relying on the programmer to manually instrument the application, which is undesirable, even though it was common practice for many researchers and prototypes. The less error-prone solution, which has recently caught up some traction, is to rely on a compiler to perform the instrumentation: the Gnu C Compiler (GCC) now implements the standard TM ABI [Ni et al., 2008] to address this. However, this results in some instrumentation of non-shared data, which cannot be proven statically as not being modified and accessed concurrently.

To better convey this transformation step, we show in Listing 2.2 the changes to the sample code shown earlier, now corresponding to the instrumented code ready to use STM. The changes introduced are analogous to those performed by GCC, which defers the logic of TM operations to the STM library — note that, in fact, the compiler will often inline the logic instead of placing the function calls that we used.

Behind this common interface, STM libraries can significantly vary the design choices of their implementation. In the following we first present some relevant design choices that differentiate contrasting STMs. Then we summarize the characteristics of several state of the art STMs that

Listing 2.2: Example of possible instrumentation applied by an STM to the sample of code shown in Listing 2.1.

```
1  bool transfer(const Account* source, const Account* dest, int amount) {
2    jmp_buf b; // used to record the instruction pointer and registers
3    setjmp(stm_buf); // record at this point so that the TM may restart the transaction back here
4    stm_start_tx(&stm_buf);
5    int tmp1 = stm_read(&source->balance); // the STM library handles the reads...
6    bool res = true;
7    if (tmp1 < amount) {
8      res = false;
9    } else {
10     stm_write(&source->balance, tmp1 - amount); // ...and the writes
11     int tmp2 = stm_read(&dest->balance);
12     stm_write(&dest->balance, tmp2 + amount);
13   }
14   stm_commit_tx(&stm_buf); // the STM commit may abort and restart to line 3
15   return res;
16 }
```

were used in the course of the work for this dissertation.

2.3.1 Update Policy

The update policy defines how the TM manages the transactional changes (i.e., updates) to shared data.

One alternative, called the lazy or deferred approach, is to keep the writes of the transaction in a separate memory region that is private to the thread executing it. This naturally ensures the desirable property that the speculative writes of the transaction are not accessible to other concurrent transactions until a successful commit is ensured. To implement this, STMs usually maintain the write addresses and corresponding new speculative values in a per-thread write-set data-structure.

In contrast, another alternative called eager or direct, applies directly the new speculative values to the actual memory address that is intended in the original non-instrumented application. However, because the transaction may still abort, some care must be taken. The previous values are backed up, in what is called an undo-set, so that they can be placed back if the transaction is aborted and the speculative writes are discarded. Furthermore, there must exist some meta-data associated with the memory regions of shared data, so that transactions can register there the fact that the associated values are still speculative and should not be accessed by concurrent

transactions.

The lazy approach is aimed at scenarios where transaction aborts and restarts are common, as in those cases the private write-set is easily discarded without changes to the shared memory, whereas the eager approach should have a very efficient commit procedure as the writes are already in-place.

In practice, implementations tend to use the lazy approach, as it turns out to be more efficient. On one hand, it concentrates the changes to shared memory in the commit procedure, which makes better use of batched messaging in the processor's bus that allows to access memory. On the other hand, the eager approach usually still needs to perform some shared memory changes during the commit, to update STM meta-data associated with each of the writes. As such, these two facts make the lazy approach more interesting, and hence why most high-performing STMs use deferred writes [Dalessandro et al., 2010, Felber et al., 2008, Dragojević et al., 2009a].

2.3.2 Conflict Detection

The detection of conflicts, between transactions accessing the same shared data (where at least one is modifying it), is enabled by the collection of the addresses read or written by the transaction into, respectively, read- and write-sets.

One possibility is to perform the conflict detection and resolution in a lazy (or also called commit-time) fashion. This means that the system detects conflicts when a transaction tries to commit, i.e., the conflict itself and its detection occur at different points in time. To do so, many STMs conduct the following verification: a committing transaction T_i must ensure that no write-set of a concurrently committed transaction T_k intersects with T_i's read- or write-set. A transaction T_k is concurrently committed with regard to T_i if it is committed after T_i started and before T_i committed.

If there is such an intersection, then it is said to exist a conflict: a read-write conflict or write-write conflict, depending on whether the intersection was detected in T_i's read- or write-set. This strategy promotes more concurrency because not all read-write conflicts are harmful to the correctness of the execution: if the transaction that is reading still commits before the writer transaction, then the resulting serialization is correct. However, by detecting conflicts lazily, this also generates situations where transactions perform many computations that are restarted,

when in fact they could had been detected earlier.

In contrast, eager conflict detection performs checks immediately during the TM operations (namely read and writes). In general, this entails modifying some meta-data associated with the addresses written (sometimes also for the addresses read, in the case of visible reads [Spear et al., 2006]), which is independent of the update policy (which may still be lazy even if the conflict detection is eager).

An interesting choice is to also consider mixed strategies: for instance, it is possible to perform eager conflict detection for writes, and lazy detection for reads. This makes sense in cases where two concurrent writes cannot be accommodated (because the TM maintains a single version for the shared memory, as most do).

Finally, some systems also vary the conflict detection granularity. Most STMs use the size of a memory word (32 or 64 bits) as the unit to keep track of in the read- and write-sets. As we shall see, in the case of HTMs, the natural unit of granularity is the cache line size of the processor. In contrast with these, STMs for object-oriented languages such as Java or C++, also adopt the object granularity, which is tightly related with programming language constructs.

2.3.3 Implementations

As a result of the ample design space for TM, the flexibility of implementing various approaches in software has led to the proposal of many STMs. In the following we mention the most prominent ones, which constitute the state of the art. We also summarize the main characteristics of these STMs in Table 2.1. Notably, we shall also use these STMs in the course of this dissertation, for which reason we delve into some of their details.

In the NOrec [Dalessandro et al., 2010] STM there is a single-global lock, which is used to regulate all the concurrency control, and is managed internally by the STM library. This lock is acquired during the commit of a transaction that contains new updates to shared memory, a so called update transaction. Furthermore, the lock also works as a logical clock, whose monotonically increasing values are used to notify concurrent transactions about updates to memory. As such, transactions keep a read- and write-set, and whenever they perform an operation in which they notice the global lock to have increased — remember that it is also a logical clock — they re-validate the read-set. This ensures that all memory locations read

Table 2.1: Categorization of three representative — albeit quite different — state of the art STMs. These STMs play an important role in this dissertation, as we often compare our contributions to them, or use them as the starting point of our new algorithms.

STM	NOrec	SwissTM	JVSTM
Correctness Criteria	Opacity	Opacity	Opacity
Progress Guarantee	Starvation-freedom	Starvation-freedom	Lock-freedom
Permissiveness	None	None	MV-Permissiveness
Update Policy	Lazy	Lazy	Lazy
Conflict Detection	Lazy	Mixed	Lazy
Conflict Granularity	Memory Word	Memory Word	Object
Programming Language	C	C++	Java

up to that point are still up to date, and, as such, the transaction is still serializable in that moment. This same validation is performed at commit before writing the updates to shared memory (while holding the global lock).

The global lock of NOrec has the advantage that commit operations perform only one expensive operation (i.e., acquiring the lock), in contrast with other STMs that may perform many atomic operations as we shall see next. On the other hand, this has the disadvantage of requiring transactions to validate the whole read-set when the global lock increases: as some memory locations were updated, the transaction must review all its reads, as there is no clue as to which locations may have been invalidated (if any). This mechanism is used to ensure the correctness guarantee of opacity.

To explore the other side of this trade-off, many other STMs [Dice et al., 2006, Felber et al., 2008, Dragojević et al., 2009a] have been designed to have a more fine-grained set of meta-data and locks, which are referred as ownership records — in contrast with the "No Ownership records" of NOrec. The idea with such STMs is to have some mapping of the addresses of shared memory to their ownership record; a practical one is to have a small number of ownership records (e.g., some thousands), and then an address is mapped to its ownership record via a hash function. In this way, the STM can register in an ownership record when the data, which is associated with it, has changed.

To have a notion of time and to order events, these STMs use similar mechanisms to that of NOrec: there is a global clock (that is not necessarily used as a lock anymore) which is incremented by update transactions, and whose value is sampled at the start of a transaction to

define the snapshot of visible shared memory. Then, a transaction T accessing shared memory will typically be able to do so without a conflict if that piece of memory maps to an ownership record that was last updated with a timestamp of the global clock that is earlier than the timestamp obtained by T to define its snapshot.

This establishes the key differences between NOrec and, for instance, SwissTM [Dragojević et al., 2009a]. Another relevant feature of SwissTM is that it uses mixed conflict detection, in which the ownership records are eagerly modified and checked during a write, but lazily (i.e., at commit-time) in case of reads. As a result, SwissTM is expected to perform better than NOrec when the application runs with a sufficiently large number of threads.

However, for large number of threads, the likelihood of having a thread de-scheduled by the operating system also increases — even more if there are more threads active in the application than cores in the machine. As such these STMs that regulate concurrency with locks fall prey of holding back progress in the system. This motivated the design of the JVSTM [Fernandes and Cachopo, 2011], which ensures Lock-freedom. To achieve this, the JVSTM uses a commit list of transactions, in which every transaction must be enqueued before committing; this list is implemented in a thread-safe way by using a CAS. The order in the list establishes the serialization order of the transactions. However, to avoid that a transaction waits for others serialized before itself, the thread executing a transaction T can help to validate and commit the transactions enqueued before T. This simple design ensures that no transaction is blocked from progressing as long as its thread is executing.

Furthermore, the JVSTM also ensures MV-Permissiveness, by maintaining multiple versions of the values written to an object over time. Then this STM erases old versions only when no transaction that is active can read that version. The snapshot of accessible versions is computed similarly as for SwissTM: there is a global clock, increased when a transaction is committed (in a lock-free way), and transactions starting sample that clock and read versions of data that are equal or older than the sampled clock. In this way, read-only transactions can always serialize at the time defined by the clock that they sample during the start, thus never aborting. In fact, all of this also means that the read-only transactions can always read the versions they need without getting blocked, meaning they are Wait-free.

This design of the JVSTM is favourable for workloads with many read-only transactions — to take advantage of the MV-Permissiveness — and with a large number of concurrent threads —

in which case the Lock-freedom is an important guarantee over Starvation-freedom. The former property results in a larger space utilization due to the multiple versions. The latter results in a more complex commit protocol for the update transactions, whose cost can be a noticeable overhead, mainly in cases in which there is not enough parallelism or de-scheduled transactions that could harm the progress of Starvation-free STMs.

2.4 Hardware Transactional Memory

The origins of TM date back to two proposals that extend the cache coherence protocol of multi-core processors [Stone et al., 1993, Herlihy and Moss, 1993]. This was the most natural way to implement the TM abstraction because the cache coherence protocol is already responsible for tracking reads and writes to memory; as such, HTMs build upon the cache coherency protocol and use its knowledge to detect conflicts and guarantee that all transactions are serializable.

In principle, HTM is the most desirable approach to implement the TM abstraction, as it heavily delegates its logic to the cache coherency protocol. This is in contrast with STMs, which tend to duplicate all that effort in the software TM libraries. By revisiting the example of code shown earlier, we now present its instrumentation in Listing 2.3 for the case of using an HTM library. The most notable aspect is that reads and writes no longer have to be instrumented. Also, as we shall see next, the implementation of the start and end procedures heavily relies on the hardware support (and naturally those procedures can be inlined in the atomic block source code). By leveraging on the cache coherence protocol, little additional logic is typically required in the HTMs, which is an important factor due to the difficulty of changing micro-processors' logic and conducting their verification [Adir et al., 2014].

However, even such a small change can take a significant effort to deploy in commercial processors. As a result, for many years most developments in HTM were conducted on simulators, the most notable being the Transactional Cache Coherence (TCC) [Hammond et al., 2004, McDonald et al., 2005] and AMD's ASF [Christie et al., 2010].

There were some notable exceptions that went beyond simulations. For instance, Azul's multi-compare-and-swap, which was used for pauseless garbage collection in a customized Java Virtual Machine in their processors, and that can be seen as a restricted form of HTM. Also Oracle, formerly as Sun, developed the Rock processor with support for HTM [Dice et al., 2012a]

Listing 2.3: Example of possible instrumentation applied by a TM library that relies on an HTM (for Listing 2.1).

```
1  bool transfer(const Account* source, const Account* dest, int amount) {
2    htm_start();
3    if (source->balance < amount) {
4      htm_end();
5      return false;
6    }
7    source->balance -= amount;
8    dest->balance += amount;
9    htm_end();
10   return true;
11 }
```

but that was cancelled before commercialization.

More recently, the maturing of TM research led to a breakthrough that changed drastically the landscape of HTM: two major market players, IBM and Intel, launched HTM support in their latest processors [Yoo et al., 2013, Wang et al., 2012a, Jacobi et al., 2012, Cain et al., 2013] (with IBM's being more aimed at high-performance computing). This represents a significant milestone for TM due to the predictable widespread availability of these processors. In particular, in the following, we overview in more detail the HTM from Intel, whose processors are substantially less expensive than IBM's (and thus more widespread).

2.4.1 Intel Restricted Transactional Memory

Intel augmented their instruction set for x86 with Transactional Synchronization Extensions (TSX) [Yoo et al., 2013], which represents the first generation of mainstream and commodity HTM: TSX is available onwards from the 4^{th} generation Core processor, codenamed Haswell, which has been deployed in millions of machines, ranging from tablets to servers.

It comprises two possible interfaces to software programs: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). The former allows to elide locks and execute code speculatively in a backwards-compatible manner. As such, its interface is designed to prefix LOCK instructions, and does not allow for software control on how to regulate the speculation. As such, for new applications written in the TM paradigm, it is much more interesting to use RTM, which exposes a conventional TM abstraction to the programmer by simply requiring the demarcation of code fragments within atomic blocks.

These extensions include the key instructions XBEGIN and XEND. This interface maps directly to the usual constructs of transactional programming by beginning and committing atomic blocks. Although the details of Intel's implementation are not disclosed, several researchers have speculated on the most likely design [Nakaike et al., 2015, Li et al., 2014, Wang et al., 2014, Viktor Leis, 2014, Karnagel et al., 2014, Yoo et al., 2013]. It is believed that Intel opted for a simple, non-intrusive design, focused on adapting the L1 cache of each core to serve as a transactional buffer. After a transaction has started, read and write accesses are placed in the L1 cache with a special bit indicating that they are transactional. When the transaction ends, the writes are atomically made visible, which means that the transactional bits are unset and this allows a snooped share request — from the cache coherency protocol — to be served. In contrast, such request would cause the abort of the transaction if it was still running: that is, in general during the transaction execution, if any memory location accessed is concurrently written, the transaction is aborted to ensure correctness; likewise for memory locations written that are concurrently accessed.

This design inherently limits RTM transactions to the size of the L1 cache. Hardware transactions are subject to abort also due to other not concurrency-related reasons, such as page faults and system calls. This clearly highlights the fact that the first generation of commodity HTM is restricted by the available hardware. In this sense, one cannot depend exclusively on RTM to implement the TM abstraction, since a transaction is never guaranteed to commit, even in absence of contention with other transactions (e.g., due to overflow of the L1 cache). This is why such HTMs are often called best-effort. Virtualizing the hardware resources for unbounded transactions is a possibility, and has been proposed already [Ananian et al., 2006], but deemed as unrealistic in the near future due to the complexity of its implementation in processors [Yoo et al., 2013, Adir et al., 2014].

As such, among the reasons for a transaction to abort in RTM, one can count page faults and interrupts, system calls, forbidden instructions, capacity overflow, and naturally conflicts in concurrent accesses to data. We summarize these in Table 2.2.

To deal with all these limitations, such best-effort HTMs require a fall-back software synchronization mechanism to be provided, so that it ensures progress in case a transaction cannot be committed with hardware support. Briefly, the idea is that upon each hardware transaction abort a predicate is queried, which decides whether to retry or to fall-back to software; in the

Table 2.2: On the left: possible bits that may be set by RTM in register EAX to indicate reasons to abort a transaction. On the right: possible events that may be monitored during RTM execution via model-specific registers.

Bit	Reason (examples)
`retry`	transient failure
`conflict`	contention to data
`capacity`	exceeded cache size
`explicit`	XABORT invoked
`other`	faults, preemption

Event	Reason (examples)
MISC1	conflicts, capacity overflow
Capacity	specifically write-set overflow
MISC3	forbidden instructions, page faults
MISC4	illegal memory accesses (I/O)
MISC5	interrupts

latter case, that alternative fall-back path must ensure also correctness in the presence of concurrent threads running hardware transactions. This means that, in fact, practical HTMs that exist in processors do not allow to implement the TM abstraction solely via hardware mechanisms, thus requiring to execute atomic blocks synchronized both with hardware and software.

The simplest fall-back approach, as suggested by Intel's optimization manual [Intel Corporation, 2012], is to use a single-global lock to protect the fall-back path for all atomic blocks. The advantage of this approach is that it is still generic, and can be implemented inside the TM library, without the programmer having to be aware of any of these concerns with the best-effort nature of the HTM.

In Algorithm 1 we show a typical implementation of a TM library, which allows to start and end an atomic block, and relies on Intel's HTM to provide the concurrency control together with the canonical single-global lock fall-back path [Yoo et al., 2013, Nakaike et al., 2015, Afek et al., 2014].

The idea is that each transaction starts with a budget of attempts in hardware (as in line 2) that is decremented upon each abort (in line 11). Note that hardware transactions started with *_xbegin()* may, at any point of their execution, roll-back automatically to the same line where it started. To distinguish between successful starts and aborted transactions the *_xbegin()* call returns a status code that can be evaluated (that is the code whose bits are set to indicate the errors of abort listed in Table 2.2).

When the budget of attempts is exhausted, the execution falls-back to the software path, and uses a single-global lock that is maintained internally by the TM library (i.e., the programmer is still using only atomic blocks in his code). For this lock to be correctly used, hardware

Algorithm 1: Intel RTM usage to provide a TM abstraction.

```
1:  HTM_START()
2:    attempts ← MAX_ATTEMPTS
3:  begin:                                          ▷ used to jump to and re-attempt with HTM
4:    htmStatus ← _xbegin()
5:    if htmStatus = _XBEGIN_STARTED
6:      if is-locked(sgl)                           ▷ ensure correctness with fall-back
7:        _xabort()
8:      else
9:        return                                    ▷ hw transaction enabled, proceed to application code
10:   ▷ hw transaction aborted; decide on what to do next
11:   attempts ← attempts - 1
12:   if attempts = 0                               ▷ decision: give up on HTM, fall-back to lock
13:     acquire-lock(sgl)                           ▷ software fall-back with a single lock
14:     return                                      ▷ software fall-back path taken, proceed to application code
15:   ▷ decision: try again to execute with hardware support
16:   goto begin

17: HTM_END()
18:   if _xtest()                                   ▷ returns true if inside a HW transaction
19:     _xend()                                     ▷ tries to commit the HW transaction; jumps to line 4 when it fails
20:   else
21:     release-lock(sgl)                           ▷ executed with lock-based fall-back
```

transactions verify that it is not locked as soon as they start (in line 6): if the lock changes concurrently after this check, then the hardware transaction gets aborted, because the lock was read transactionally and so the hardware detects any concurrent conflicting (i.e., write) access to it.

Recently, it has been proposed to verify (i.e., subscribe) the lock lazily, before the commit of the hardware transaction [Calciu et al., 2014b]. However, it is still open to debate whether that approach is generalizable, as there are corner cases in which that can lead to an incorrect execution [Dice et al., 2014a].

This eager subscription is required because a thread executing in the software fall-back path has no monitoring in place that prevents inconsistent reads (i.e., no instrumentation). Consider for instance the example shown in Figure 2.1 containing an application which preserves the invariant $x = y$, and a thread that executes in the fall-back path and reads $x : 0$. If one allows a concurrent hardware transaction — which does not eagerly subscribe to the lock — to write $x = y = 1$ and commit, then the fall-back path thread could read $y : 1$ inside the atomic block and violate the invariant. With the verification in line 6, pessimistic executions are safeguarded from inconsistencies because hardware transactions abort preemptively to avoid any such hazard.

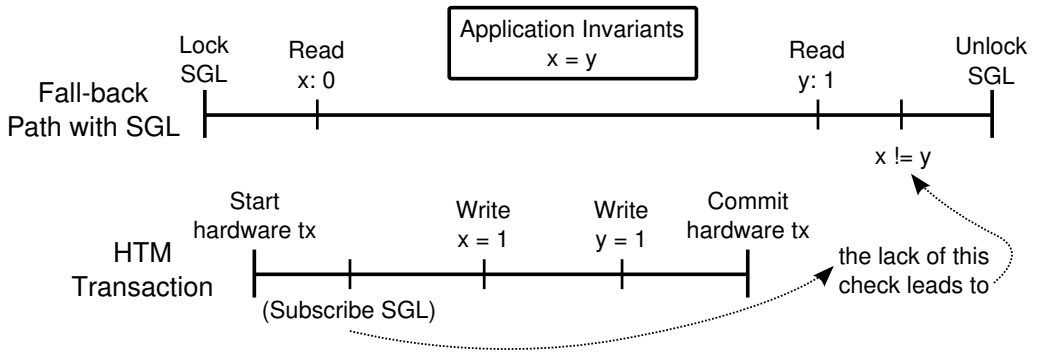


Figure 2.1: Illustration of the relevance of subscribing to the single-global lock (SGL) when it is used as the fall-back of an HTM. This concurrent execution, of a hardware transaction and a fall-back path with the SGL, shows that the lack of the subscription (omitted in parenthesis) can lead to an invariant being broken in the fall-back path of the atomic block of the application (i.e., to inconsistent reads in general).

2.4.2 Other Hardware Implementations

IBM has released several other HTMs in their most recent processors. While all them share the same best-effort nature of Intel's RTM, IBM chose to change a handful of other design choices, which we now briefly describe. Quite interestingly, IBM did not release a single HTM implementation, but instead made three different implementations in their processors.

The Blue Gene/Q [Wang et al., 2012a] HTM offers two alternative modes to keep track of transactional accesses. The short running mode performs this tracking in the L2 cache, meaning that all accesses must bypass the L1 cache, thus adding some latency to the execution in transactional mode. In contrast, the long running mode allows to monitor the transactional accesses in the L1 cache, but for that to be possible the processor first evicts that cache before starting the transaction. As such, there is trade-off in that it allows faster memory accesses at the cost of a constant overhead in the transaction begin. As the L2 cache is shared by all cores, the accesses of a given transaction are marked by a corresponding transaction identifier, which is acquired automatically when the transaction starts. There are limited identifiers managed by the processor, which are periodically reclaimed, but whose absence of availability can block the transaction start (e.g., because the transactions are very short and the reclamation period is large).

The zEC12 [Jacobi et al., 2012] HTM works similarly to Intel's HTM, as it also marks the

L1's cache lines with read or write bits for transactional accesses, but the actual write values are stored in a specific purpose cache. This HTM has the unique property of having constrained transactions whose success is guaranteed (unless a data conflict happens or forbidden instructions are found). It is, however, quite constrained: up to 32 instructions and 256 bytes of memory accessed.

Finally, the Power8 [Adir et al., 2014] HTM uses another specific purpose cache that is linked to the L2 cache. The specific purpose cache records the transactionally accessed addresses, although the actual values are still stored in the L2 cache. As such, this specific cache serves the purpose of the extra bits that other processors use for the tracking. A noteworthy feature of the Power8 HTM is that it provides additional instructions to suspend a transaction and to resume it. This allows to execute non-transactional operations, although one should bear in mind that the suspend and resume operations have a constant overhead associated, which is roughly on the order of magnitude of starting and committing a transaction. This is in contrast with AMD's ASF [Christie et al., 2010] proposal that allowed individual non-transactional operations to be prefixed as such in the scope of a transaction. This Power8 HTM also provide the new type of roll-back only transactions: they do not track reads, hence they do not guarantee isolation. Yet, they still provide failure atomicity and their usage has been proposed in the domains of debugging [Le et al., 2015], fault-tolerance [Kuvaiskii et al., 2016] or as alternative fall-back paths [Felber et al., 2016].

2.5 Hybrid Transactional Memory

The pros and cons of both HTMs and STMs make them a perfect match. HTMs execute transactional code with higher performance, but their nature is quite restricted as to which transactions may succeed. STMs add some overhead to transactions, but are far more flexible in terms of completing transactions successfully. Researchers recognized this synergy and proposed Hybrid TM (HyTM) implementations.

As such, the idea is to execute concurrent transactions both in hardware and software, so that we can effectively exploit the benefits of each of the underlying implementations: if a hardware transaction fails often, then it is preferably executed with the STM, without stopping concurrent hardware transactions as in the case of the single-global lock fall-back.

However, the difficulty is in coupling both concurrent hardware and software transactions, such that their respective concurrency control mechanisms can detect the actions of one another. This can be solved as proposed by PhasedTM [Lev et al., 2007], in which all transactions execute either in software or hardware. This has the advantage of simplifying the logic to couple both modes, but the disadvantage of a costly procedure to switch modes (that is, a stop the world approach). As such, it is desirable to be able to run both HTM and STM transactions concurrently.

The inherent challenge of such a HyTM, in which transactions run simultaneously in both modes, is that the HTM side is usually agnostic of the presence of the STM transactions — and while it may be quite feasible to change the STM implementation to be aware of the HTM, that is not the case with hardware. As such, it is difficult to prevent a hardware transaction from committing when it should not. The intuition is that a software transaction may be executing the commit procedure, and it may have already been deemed as successful; yet, it is always possible for a new hardware transaction to execute, commit, and invalidate the software transaction, because its concurrency control is independent from the STM.

Therefore most HyTMs end up instrumenting the code that will run inside hardware transactions too, in order to allow for the detection of conflicts with concurrent software transactions (e.g., [Damron et al., 2006]). This has the clear disadvantage of slowing down the supposedly fast-path of the hardware transactions in the HyTM.

In the following sections we present the most recent and notable exceptions to this claim, which have given some prominence to HybridTMs in the scope of the recent best-effort HTMs.

2.5.1 The Hybrid NOrec Implementation

The NOrec STM [Dalessandro et al., 2010], which was presented earlier in Section 2.3.3, is mostly known for its simple design that requires a small amount of software instrumentation in the application. As a result, its potential for an efficient HyTM is very high, as it should be possible to implement its interaction with the HTM also in a simple way that involves NOrec's global lock. Indeed, as we shall see, this is not very different from HTM's canonical single-global lock fall-back path.

Dalessandro et al. used this strategy when proposing HybridNOrec [Dalessandro et al., 2011].

Some of HybridNOrec's proposed optimizations, however, were designed for AMD's ASF proposal [Christie et al., 2010], which has not been commercialized, in which the non-transactional loads and stores were used to reduce hardware transaction spurious aborts generated by the coupling of the STM and HTM. These optimizations are thus not available in implementations relying on the HTMs available from Intel and IBM.

The base idea of HybridNOrec is quite simple: to replace the typical software fall-back path, which relies on a single-global lock for synchronization, with the NOrec software transactions. To ensure correctness between concurrent hardware and software transactions, it is necessary to have a mechanism similar to the lock subscription of pure HTM (as in line 6 of Algorithm 1). The idea of HybridNOrec, which we summarize here, is to use NOrec's single-global lock to manage that interaction: hardware transactions are successful only when NOrec's lock is not taken. As we now may have hardware transactions running concurrently, we need to ensure that they are safeguarded from the non-atomic writes of software transactions, which are published during the write-back that takes place under NOrec's commit procedure. So this is a very similar concern to that explained earlier for the interaction of the single-global fall-back and hardware transactions.

We illustrate this in Figure 2.2, between transactions Tx_1 and Tx_2, where the hardware Tx_2 gets aborted because the NOrec single-global lock is acquired concurrently. This prevents hazardous situations for Tx_2, such as the inconsistent reads that are shown in the illustration and that would happen if the lock had not been subscribed by the hardware transaction.

Note that this constrains concurrency less than the typical single-global lock used with HTM, because NOrec's single lock is taken only during the commit of a software transaction (and not during the whole transaction); hence there is more room for parallelism between hardware and software transactions as the single lock is held for smaller periods of time. Furthermore, read-only transactions in NOrec do not need to acquire NOrec's global lock.

So far we have seen how NOrec's software transactions notify each other of new commits by incrementing the global clock, which also serves to protect hardware transactions from observing inconsistent states that do not generate aborts (such as the case hypothesized in Figure 2.2). It is also the case that hardware transactions perform a similar task: namely, they warn software transactions of the fact that a hardware commit has occurred, and that a read-set re-validation is needed. For this, hardware transactions increment a separate counter, whose value is tracked also by software transactions (similarly to the global clock). The motivation to use a separate counter

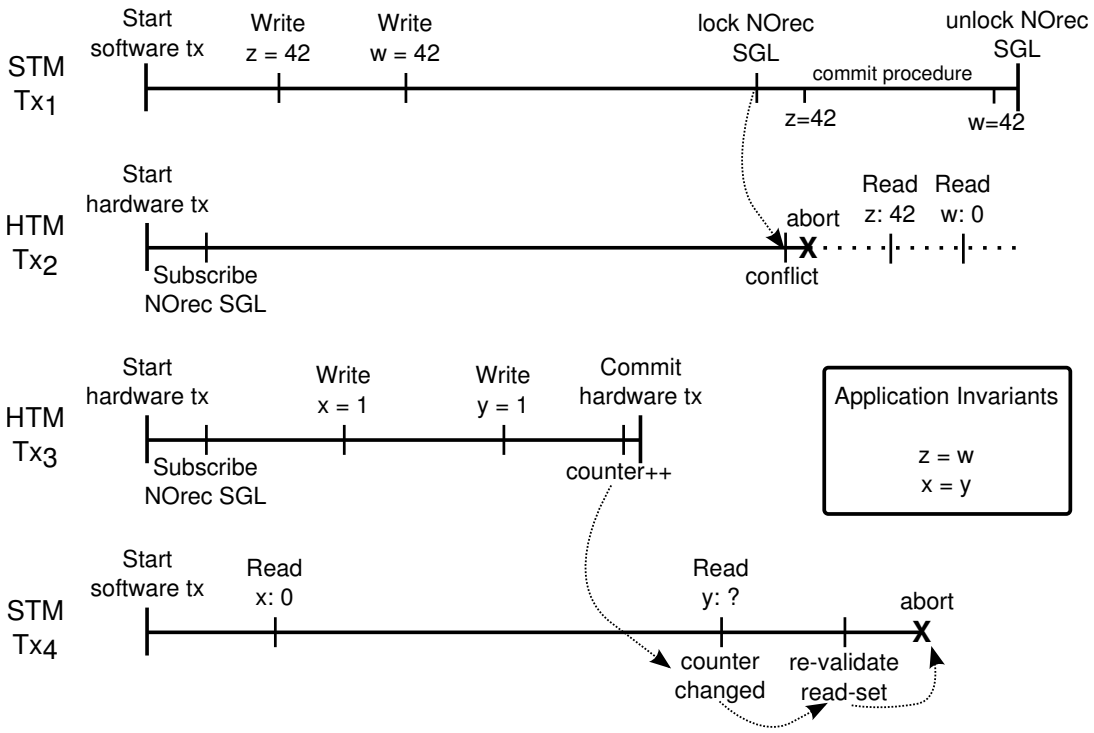


Figure 2.2: Illustration of the relevance of HybridNOrec's subscription mechanisms to ensure correctness: NOrec's *SGL* is used to notify software and hardware transactions of software commits (as the case for Tx_1 with respect to Tx_2), and NOrec's *counter* is used to notify software transactions of hardware commits (as the case for Tx_3 with respect to Tx_4).

is that hardware transactions subscribe to the global clock right after they start, so incrementing a different counter in the end avoids many spurious conflict aborts between hardware transactions (that would happen with respect to the global clock).

We illustrate this in Figure 2.2 between transactions Tx_3 and Tx_4. The hardware transaction Tx_3 atomically publishes its writes for the application variables together with an increment of NOrec's counter. This allows the concurrent software transaction Tx_4 to detect that something has changed, by keeping track of NOrec's counter value that was just changed, and thus to re-validate its read-set and find a data conflict. Otherwise, Tx_4 would read an inconsistent value for y with respect to that of x, violating Opacity and an application invariant, which could lead to unexpected results.

Furthermore, hardware transactions keep a thread-local flag to register whether they have performed some write to shared memory. This entails that write operations are instrumented to set that flag to true. If the flag is false, then the hardware transaction need not increment any

counter, as its read-only commit shall be harmless to concurrent software transactions.

Another interesting optimization is to have software transactions announcing their presence, by incrementing a counter when they begin, and decrementing it when they finish (with atomic operations). This allows a hardware transaction to check whether there is any software transaction alive. If not, then the hardware transactions need not increment any counter as well. Note that this check is performed within a hardware transaction, so it is similar to the lock subscription: if some software transaction appears after the check, it aborts the hardware transaction with a conflict.

2.5.2 Reduced Hardware Transactions

Typical HyTMs, such as the HybridNOrec presented above, either execute atomic blocks with the HTM or with the STM in case the first one fails often. The key insight for Reduced Hardware Transactions [Matveev and Shavit, 2013] is that there is an in-between these two extremes: if a hardware transaction fails, we may still use the HTM to help the subsequent fall-back software transaction to execute with less instrumentation, by relying on the hardware capability of tracking concurrent conflicts. However, this is only useful if we can use the HTM in such a way that it generates smaller hardware transactions, which are less likely to trigger the several limitations of HTM that cause aborts.

As such, the Reduced Hardware NOrec [Matveev and Shavit, 2015] HyTM proposes to use: a fast-path with hardware transactions having minimal software instrumentation; a mixed-path that uses the STM but that executes parts of the software transaction protected by the hardware transaction (thus allowing to remove part of the software instrumentation); and a canonical slow-path that uses the full STM as described for HybridNOrec above.

Hence, the term reduced comes from the fact that the fall-back will use the HTM for smaller portions of code, and hopefully succeed more often. Naturally, there is always the case in which even that mixed fall-back will cause the HTM to abort, in which case it resorts to the (typical) slow fall-back.

The implementation of Reduced Hardware NOrec thus introduces this new mixed-path, which consists of adding two hardware transactions to the mixed path. The first one, named prefix, encapsulates the read-only initial part of the atomic block (which may not exist, if the

atomic block begins right away by writing to the shared memory). As such, this allows to rely on the HTM capability of conflict detection, instead of having the reads instrumented for STM execution. Naturally, this may lead to capacity aborts due to too many reads, for which reason the size of this prefix hardware transaction is adapted. Right before the prefix transaction is committed, the global clock of NOrec is sampled to obtain the snapshot of the subsequent software NOrec transaction that follows.

Due to the adaptivity of the prefix transaction size, an atomic block being executed in the mixed-path may then run some reads with STM instrumentation.

The second hardware transaction that is added to the mixed-path, named postfix, encapsulates the atomic block from its first write on until the end. This means that all writes are encapsulated in a hardware transaction, thus preventing concurrent threads from witnessing its intermediate states. As such, this allows hardware transactions to subscribe lazily to the global clock, i.e., right before they commit, in cases where there are no slow-path transactions running. This can be verified similarly to the previous optimization described for HybridNOrec to track software transactions that are active. Hence, in the case that transactions fall-back only to the mixed-path, then hardware transactions can benefit from this optimization of subscribing lazily, which improves concurrency between software and hardware transactions.

In summary, this technique allows to rely more on HTM, even in cases where its obvious usage would result in aborts stemming from the hardware limits. Note, however, that managing these paths entails placing verifications on the TM library to execute the correct logic for each situation. Namely, the fast-path must not only synchronize with the mixed-path, but also with the slow-path. As such, situations in which the slow-path is taken, even if rarely, can quickly degenerate the executions to tend towards the slow-path instead of using the mixed-path. Ideal workloads, where most improvements are observed, are those where the mixed-path can be used effectively and without needing to execute in the slow-path.

2.6 Distributed Transactional Memory

While the focus of this dissertation is on the TM paradigm for developing concurrent applications in multi-core settings, during the past recent years we have also witnessed significant changes in the development of distributed applications. The advent of the cloud computing

paradigm has empowered programmers with the ability to scale out their applications easily to hundreds of nodes. However, developing applications capable of effectively exploiting the computational capabilities of large scale distributed cloud platforms is, similarly to that of concurrent applications, far from being a trivial task.

In this sense, the abstraction of TM with serializable (and distributed) transactions is equally important to help programmers harness the power of large distributed systems, playing an analogous role to that of TMs in multi-core machines. Indeed, several distributed data platforms can be seen as Distributed TMs (DTMs) as they provide distributed serializable transactions, with shared data maintained in-memory, often implemented through optimistic concurrency control techniques and without requiring the programmer to pre-declare the data accessed in the transactions (i.e., no need for static transactions). Therefore, we briefly mention only some relevant works in the area of DTMs, as the most significant part of our contributions are not focused on DTMs.

In DTMs a group of servers maintains (and possibly replicates) data, to which clients perform requests to execute transactions. Servers mostly synchronize only at the end of a transaction, during the commit, to either atomically update data across the servers that replicate it or to abort the transaction. From a high-level perspective, the steps followed are analogous to a centralized TM, as described earlier. The key difference is that network communication is involved in some of the steps, which entail higher costs, and thus may need to be performed with caution. Furthermore, fault-tolerance needs to be ensured.

One key advantage of DTMs is that they preserve scalability and performance despite being fault-tolerant, when compared with traditional approaches: in state-machine replication, every update transaction is executed in every server [Schneider, 1990], and in primary-backup replication every update transaction is executed by the primary server; as such, throughput is limited by the processing power of a single replica. In DTMs different replicas can process different transactions, and thus provide scalable performance while remaining fault-tolerant. Despite their differences, there have been proposals that exploit jointly DTMs with state-machine replication, so that some transactions can be executed deterministically/pessimistically for progress guarantees and coping with irrevocable actions [Kobus et al., 2013].

We distinguish two types of DTMs with regard to their replication technique: fully replicated DTMs in which every replica holds a copy of all shared memory, and partially replicated DTMs

in which each replica holds a copy of a subset of all shared memory. In fully replicated DTMs every access to the distributed shared memory can be processed locally during the transaction execution, whereas in partially replicated DTMs this may entail network communication to other replicas (which may be reduced with proper data placement [Paiva et al., 2013]). In contrast, in fully replicated DTMs, updates produced by transactions must be committed in every replica, whereas partially replicated DTMs reduce this to a subset of the replicas — this cost is significant as it implies a consensus operation (such as total order primitives [Vale et al., 2013]). As such, partially replicated DTMs typically perform worse for a small number of replicas [Hirve et al., 2014], but have the potential to scale much more if provided with proper data placement [Schiper et al., 2010, Peluso et al., 2012c, Peluso et al., 2012b].

Dependability in fully replicated DTMs is typically ensured by resorting to a group communication primitive to trigger the commit procedure in the replicas: for instance, several proposals use Atomic Broadcast [Sciascia et al., 2012, Couceiro et al., 2009, Kotselidis et al., 2010] so that concurrent commits are totally ordered and each replica can process them in parallel, with a deterministic outcome. In general these protocols work similarly to centralized STMs, as reads are served locally, writes buffered locally, and the validation is processed independently at each replica after receiving the total order delivery.

DTMs typically employ multi-versioned concurrency control [Bernstein et al., 1987] because it allow efficient read-only transactions, by sparing them from any aborts and remote validations, as they can serialize in the past by reading old versions. This is analogous to the multi-versioned STM described in the JVSTM implementation in Section 2.3.3. However, for DTMs, any meaningful amount of update transactions in the workload shall prevent scalability in fully replicated DTMs despite the efficiency of read-only transactions. To try to overcome the bottleneck on the total order delivery across all replicas, several systems have exploited optimistic total order delivery [Hirve et al., 2014, Carvalho et al., 2010, Peluso et al., 2012a]. They exploit the fact that the first phase of the group communication protocol often delivers messages in the same order as their final total order. Hence, they speculatively proceed with the validation and commit earlier than the total order delivery, overlapping this processing with the network communication. Naturally, this comes at the cost of potentially having to fix the commit order later.

As mentioned above, partially replicated DTMs take a different approach to enhance scalability: there have been several proposals that explore a key property called *genuine* partial

replication, according to which the execution of a transaction can only involve nodes that replicate data items it accessed [Schiper et al., 2010]. This property is of the utmost importance to enable high scalability in update-intensive workloads, as it rules out non-scalable solutions based either on centralized components (which may turn into bottlenecks/single points of failure) or on full-replication (which induces unacceptable overheads to propagate updates across the entire system).

More recently, GMU [Peluso et al., 2012c] and SCORe [Peluso et al., 2012b] were the first proposals to combine the following techniques to provide scalable DTMs: read-only transactions benefiting from abort-freedom due to the availability of multi-versions, and update transactions using a genuine protocol with partial replication. SCORe provides 1-Copy Serializability [Adya, 1999], whereas GMU provides a slightly more relaxed criteria, called Update Serializability [Adya, 1999]. This difference stems from trade-offs in the protocol design: GMU uses a vector clock protocol in which read operations can exploit the benefit of caching, whereas SCORe uses a single distributed clock abstraction. This scalar clock in SCORe reduces its book-keeping and communication overheads with regard to GMU. On the other hand, by relying on a vector clock, GMU can track conflict information at a finer granularity than SCORe. This allows GMU to avoid spurious aborts that are instead incurred in by SCORe, and also to provide stronger guarantees of accessing fresher data than with SCORe.

2.7 Benchmarks and Applications for Transactional Memory

To evaluate the contributions developed in the course of this dissertation, we have relied on existing benchmarks and applications, which have been developed using atomic blocks and the TM abstraction. Typically, these execute with a configurable number of threads, each of which executes transactions. The fundamental metric that we use is then to measure how much faster a concurrent execution is, when compared to a sequential one, which is often mentioned as speedup.

In the next sections we briefly summarize each of these benchmarks and applications. We also list them in Table 2.3 [1].

[1]The number of lines of code reported is with respect to our C/C++ implementations. Some of these benchmarks were also ported to Java to evaluate some of our contributions developed in that programming language.

Table 2.3: TM applications used in our evaluation. These 15 benchmarks span a wide variety of workloads and characteristics.

Benchmark	Lines of Code	Atomic Blocks	Description
Data-Structures	3702	12	Concurrent Red-Black Tree, Skip-List, Linked-List and Hash-Map with workloads varying contention and update ratio.
STAMP [Minh et al., 2008]	28803	35	Suite of **8** heterogeneous benchmarks with a variety of workloads (genomics, graphs, databases).
STMBench7 [Guerraoui et al., 2007]	8623	45	Based on OO7 [Carey et al., 1993] with many heterogeneous transactions over a large and complex graph of objects.
Memcached [Spear et al., 2014]	12693	120	Caching service with many short transactions that are used to read and update the cache coherently.
TPC-C [TPC Council, 2011]	6690	5	OLTP workload with in-memory storage adapted to use one atomic block encompassing each transaction.

2.7.1 Concurrent Data-Structures

Throughout our work we consider a red-black tree, a hashmap, a skip-list and linked-list, as examples of concurrent data structures, which represent important building blocks of parallel applications.

These data-structures have two interesting characteristics: they are very hard to parallelize efficiently using locking schemes, and they are generally challenging for STMs as they tend to generate small transactions that suffer from relatively large instrumentation overheads.

Traditionally, many TM proposals have been evaluated with some of these data-structures (e.g., [Hammond et al., 2004, Dragojević et al., 2009a, Felber et al., 2008, Dalessandro et al., 2010]. We based our implementations on those of Synchrobench [Gramoli, 2015], a recent work that studies the performance of TM algorithms (among others) in concurrent-data structures.

These data-structures are also very flexible as they are highly parametrizable and allow to generate various degrees of contention between concurrent threads. Each data-structure has three atomic blocks in its source code, corresponding to the operations to insert an element, delete an element, and verifying whether an element exists. This last operation is executed as a read-only transaction, which can be optimized in the case of STMs, but is not relevant for

HTMs.

2.7.2 The STAMP Suite

The STAMP suite [Minh et al., 2008] is a popular set of benchmarks for TM, encompassing applications that are representative of various domains and that generate heterogeneous workloads. Each benchmark, among the 8 available, is quite homogeneous and has limited flexibility to produce different workloads. However, the fact that they are all quite different, generates an interesting suite of benchmarks. These benchmarks are ports of existing applications, or emulations of typical tasks in different areas of computing (namely, Genome for genomics, Intruder for intrusion detection, Labyrinth for circuitry routing, Vacation for online shopping, SSCA2 for graph analysis, KMeans for clustering, and Yada for mesh refinement).

As a result, these benchmarks have a number of atomic blocks that varies from 3 to 10. Some tend to have small transactions (Intruder, KMeans and SSCA2), while the rest have larger transactions (that are less favourable to best-effort HTMs). Some are well suited for larger concurrency degrees by having small contention levels between transactions (Genome, KMeans and SSCA2), while the rest tends not to scale beyond some point because of too many conflicts.

We note that the STAMP benchmarks do not have read-only transactions identified in the source code.

2.7.3 STMBench7

STMBench7 [Guerraoui et al., 2007] is a highly customizable benchmark, with three different standard workloads that vary the percentage of read-only transactions: read-dominated (90%), read-write (60%), and write-dominated (10%). Besides that, we may also control a series of other parameters such as one that includes long, highly-conflicting transactions in all the workloads. This benchmark is an adaptation of the OO7 benchmark [Carey et al., 1993] for object databases, which makes it a nice fit for the new setting of in-memory data.

STMBench7 implements a shared data structure, consisting of a set of graphs and indexes, which models the object structure of complex applications. It supports many different operations, varying from simple to complex. Both short traversals and short operations access a small part of the graph of objects, most of the time using the indexes to short cut the path. These operations

are very fast, executing on average under one millisecond on a modern processor. On the other hand, long traversals sweep most of the graph of objects. The execution of one of these traversals is in the range of hundreds of milliseconds.

As a result of all this variety, the STMBench7 ends up having 45 different atomic blocks in the source code, which is a sign of its complexity and size.

2.7.4 Memcached

Memcached is a popular object caching service that is often placed in front of application servers powering web applications. It is notably known for its adoption at a massive scale inside Facebook [Nishtala et al., 2013]. Each Memcached instance works as an in-memory store, with limited storage space, which leads to the usual eviction procedures that govern caching policies. As a result, there is often contention in accessing the data between lookup operations (that are very frequent) and updates (that can generate more than one modification due to the evictions, i.e., deletions).

This application has been widely deployed in its original form, which relies on locking to correctly synchronize all the accesses. Thanks to a recent effort, Memcached has been ported to have atomic blocks instead, and thus rely on a pluggable TM library [Spear et al., 2014]. As a result, this became a realistic benchmark with 120 atomic blocks in its source code.

2.7.5 TPC-C

TPC-C is a well known benchmark to evaluate online transaction processing in databases [TPC Council, 2011]. Its implementations follow a specification that was devised to represent realistic operations. It mimics the activities of a whole-sale supplier, for instance similar to Amazon, which owns different warehouses with corresponding products, stocks and customers. The benchmark includes five possible transactions, which allow customers to purchase and pay for items and the warehouse to deliver them, as well as checking up on available stock levels and the status of an order.

We have ported an in-memory sequential implementation of TPC-C, which was written for evaluating H-Store [Kallman et al., 2008], by protecting the TPC-C transactions with atomic blocks that are then implemented by a pluggable TM library.

3 Comparison Study of Synchronization Techniques

During the past decade we have witnessed many proposals in the area of TM. Most of the early ones focused on producing STMs, whereas in the most recent years there has been a gradual shift to HTMs (and HyTMs also), given the availability of commodity processors with support for HTM. As we have hinted in Chapter 2, despite the availability of recent HTMs, their best-effort nature allows STMs to still play a relevant role when it comes to choosing which one is best for a given workload.

Furthermore, for many years, locking has represented the *de-facto* standard approach to synchronization in concurrent applications. However, as we argued for earlier, locks are associated with an inherent complexity and error-proneness when aiming for high performance [Pankratius and Adl-Tabatabai, 2011]. In fact, this is part of the motivation behind the intense research on alternative methodologies, which has resulted in the paradigm of TM.

As such, we are posed with several questions with respect to the performance attainable by TM, in particular when considering the novel processors with HTM, and in comparison also to the more traditional approach with locks. In particular, how competitive are available HTMs when compared with state of the art STMs? Will the performance of HTM be sufficiently alluring to turn TM into a mainstream programming paradigm? What role will STM play now that HTM is so widely available? How limiting are the architectural restrictions of existing best-effort HTM designs?

These questions motivated us to conduct a comparative study of different synchronization techniques for concurrent applications. In the following section we provide an overview of our approach and results observed.

3.1 Overview

To answer the previous questions, and to make a comparative assessment of the state of the art that will help outline the rest of the work in this dissertation, we conducted the largest study on TM-based synchronization to date.

We compared, from the two-fold perspective of performance and energy-efficiency, a range of synchronization mechanisms: 6 lock based approaches with different granularities; 4 state of the art STMs; 1 HTM; and 2 HyTMs. We used several of the TM benchmarks and applications presented in Section 2.7, resulting in a highly heterogeneous set of workloads, encompassing 1) the STAMP suite of benchmarks, 2) the in-memory caching system Memcached, and 3) concurrent data structures that are widely used as building blocks of parallel applications (yet, hard to parallelize efficiently).

While our study addressed both metrics of performance and energy-efficiency, which is something that had not been done in general in the scope of TM, we found that most of the time there is a high correlation between them. That has also helped to steer the direction of the remaining of this dissertation, as we later focus on improving performance, with the side-effect of equally improving energy-efficiency.

The remaining results of our study allowed us to draw two main conclusions:

Lights and shadows for HTM: Approaches based on Intel RTM yielded outstanding performance in workloads characterized by small transactions, such as concurrent data structures and Memcached, but only with two of the STAMP benchmarks. RTM performance is strongly dependent on the access patterns to L1 cache, and long running transactions can lead to frequent cache capacity exceptions and spurious aborts. When transaction-intensity is medium (i.e., contention and transaction length are up to medium levels), RTM is only the best choice for a limited degree of parallelism, and it is slightly better on the energy side than on the performance side. The impact of its hardware limitations are highlighted by several STAMP benchmarks that generate long transactions, and in which RTM is outperformed by both locking and STM solutions. On the other hand, RTM shines as a synchronization primitive for concurrent data structures, for which it is by far the best choice in all considered workloads, with speed-ups up to 3.3× over the best alternative scheme.

STM is still competitive: Our study also showed that STM is quite competitive as an all-around solution across benchmarks, workloads, and parallelism degrees. Its evolution throughout a decade of intense research has resulted in several highly-optimized mechanisms, which achieve performance comparable to that of fine-grained locking. This does not mean that STMs embody a perfect solution; instead, this result highlights the current limitations of HTM support, which still allow STM to be the most robust solution to date.

Further, the results of our study unveiled a number of critical issues related with HTM performance and allow for identifying several research problems, whose timely solution could significantly enhance the chances for HTM to turn into a mainstream paradigm for concurrent programming:

HyTMs: a missed opportunity? The outcome of our study for what concerns the efficiency of HyTMs, when employed in conjunction with Intel's HTM, is rather grim. The mechanisms currently adopted to support the simultaneous coexistence of HTM and STM induce high overheads in terms of additional spurious aborts. Our study highlighted that these costs make HyTM generally less efficient than solutions based purely on STM or RTM with a locking-based fall-back. This motivates further research in the design of architectural support (e.g., non-transactional loads and stores in the scope of hardware transactions) capable of exploiting the potential synergies of HyTMs.

Complexity of HTM tuning. HTM performance can be significantly affected by the settings of several parameters and mechanisms. Without proper tuning, Intel's RTM suffered average throughput losses of 72% and of 89% in power consumption. Also, the optimal configuration of these parameters can vary significantly, depending on the characteristics of the workload. These findings urge for novel approaches capable of removing from the shoulders of programmers the burden of manually tuning HTM, by delegating this task to runtime or compiler based solutions.

Relevance of selective instrumentation. When using HTM, and also STMs with compiler-based automatic instrumentation, the TM library ends up tracing every memory access performed within a transaction. We showed that this can cause some increases in the transaction footprint's size, amplifying the instrumentation overheads in STM, and the chances of incurring in capacity exceptions in HTM. These results motivate for research on cross-layer mechanisms operating at the compiler and at the hardware level, aimed to achieve selective instrumentation in a way that

is both convenient for the programmer and efficiently implementable in hardware.

In the following Section 3.2 we discuss specific works that have performed similar comparative studies. Then, in Section 3.3 we present preliminary information to help understand the techniques used in the study. We then present our study in Sections 3.4-3.5. In Section 3.6 we identify several research questions suggested by the findings of our study. Finally, Section 3.7 summarizes this chapter.

3.2 Related Work

Often in the past we have seen many TM proposals that evaluate a novel algorithm by comparing it with one or two previously well established TM systems in a small set of benchmarks. That is the case for most the STMs overviewed in Section 2.3 (e.g., [Felber et al., 2008, Dragojević et al., 2009a, Dalessandro et al., 2010]), but also for the recent HTM implementations released in Intel [Yoo et al., 2013] and IBM processors [Wang et al., 2012a]. This study aims at addressing the lack of a fair and uniform comparison among the broad range of alternative implementations proposed so far in the vast literature on TM.

A notable exception to this gap was published concurrently to our work [Goel et al., 2014], whose coincidental timing stems from the recent availability of the Intel processors with HTM, which spurred this interest in establishing common grounds for comparing it with existing work. In common, both our work and that of Goel et al. compare the performance and energy expenditure of Intel RTM against that of an STM. On top of that, our work considers also other STMs, HyTMs, and fine-grained locking approaches. Furthermore we test more benchmarks and perform also a study to hand-tune the RTM-based approaches.

Another interesting work, which was published after our study, was conducted by IBM [Nakaike et al., 2015]. That study focused only on HTMs, and compared the performance of the three HTMs of IBM and that of Intel. In contrast, another more recent work focused exclusively on studying the impact of contention management in the energy efficiency of STMs [Issa et al., 2015].

Besides these two concurrent or more recent works, the concern for both performance and power consumption metrics had only been marginally explored in the scope of TM, and mostly relying on simulation studies that did not target Intel's architecture (whose internals are only

partially disclosed). In both [Gaona et al., 2013,Ferri et al., 2010] the authors assess the behaviour of different HTM implementations via simulation (the latter focusing on embedded systems). The approach was also taken by [Baldassin et al., 2009], where the power consumption of one STM was studied via simulation. More recently, [Gautham et al., 2012] studied both power consumption and performance in a non-simulated environment. Yet, this work considered a restricted set of synchronization alternatives focusing mainly on one STM.

Before the recent release of Intel processors with HTM, researchers had already proposed some theoretical improvements to best-effort HTMs [Afek et al., 2013,Matveev and Shavit, 2013]. We integrate these mechanisms in our HTM-based approaches, together with the optimizations described in the scope of HybridNOrec in Section 2.5.1, to ensure that their performance is maximally tuned.

Traditional lock-based synchronization techniques have been thoroughly studied for decades. In [Ferri et al., 2009], the authors show that the power consumption of locking primitives can be improved by exploring a trade-off between processor deep sleeping states, frequency downsizing and busy waiting. We highlight a recent work [David et al., 2013], which studied the impact in performance of different lock designs and hardware architectures (without however considering TM).

3.3 Preliminaries for the Study

In the following sections we present several details of the synchronization techniques used. We start, in Section 3.3.1, by describing the types of synchronization techniques considered. We then explain the methodology used in the study, in Section 3.3.2. Finally, we provide a preliminary study that showcases the limitations of the best-effort nature of Intel RTM in Section 3.3.3.

3.3.1 Synchronization Mechanisms Considered in the Study

In this comparative study we considered the following different synchronization mechanisms, which are also listed in Table 3.1, and most of which have been presented in detail in Chapter 2:

Locks — Decades of research on lock-based synchronization have resulted in a plethora of different implementations, many times trading off subtle changes with great impact in performance.

Table 3.1: Synchronization mechanisms compared in our study.

Mechanism	Description
Locks	Coarse/fine-grained locking [David et al., 2013]: TTAS, Spin, RW (pthreads), MCS, CLH, Ticket
STMs	TL2 [Dice et al., 2006], TinySTM [Felber et al., 2008], SwissTM [Dragojević et al., 2009a], NOrec [Dalessandro et al., 2010]
HTMs	RTM-GL [Yoo et al., 2013] (global lock), RTM-FL (fine locks)
HyTMs	RTM-TL2 [Matveev and Shavit, 2013], RTM-NOrec [Dalessandro et al., 2011]

We consider 6 different lock implementations [David et al., 2013] and apply both coarse and fine-grained locking strategies. Contrarily to the other approaches employed in the study, fine-grained locking requires a *per-application* lock allocation strategy, which is a non-generalizable and error-prone task [Pankratius and Adl-Tabatabai, 2011].

STM — With STM, reads and writes to shared memory (inside atomic blocks) are instrumented to detect conflicts between transactions. This instrumentation induces overheads that can have a detrimental impact on the efficiency of STMs. Yet, much research has been devoted over the last years to reduce STM's overheads. For our study we selected four state of the art STMs, which are representative of different choices in the design space of TM. These include an STM optimized for validations at commit-time (TL2 [Dice et al., 2006]); to maximize performance at low thread counts (NOrec [Dalessandro et al., 2010]); to maximize scalability (TinySTM [Felber et al., 2008]); and to make a fair choice of aborts between long and small transactions via contention management (SwissTM [Dragojević et al., 2009a]).

HTM — As we have seen, HTM implements a concurrency control scheme in hardware, avoiding the overheads of STM instrumentation. In this study, and in the rest of this dissertation, we use Intel RTM as our HTM implementation, greatly motivated by its ubiquity in modern processors. For one of the approaches, we use it in co-operation with the single-global lock fall-back path (RTM-GL). An obvious extension of this idea is to use fine-grained locks in the fall-back path (RTM-FL). As the TM abstraction is motivated by the need of relieving programmers from the complexity of designing locking schemes, the usage of fine-grained locks as a fall-back for HTM sounds somewhat contradictory. However, this choice allows us to assess to

what extent a simplistic fall-back (using a single lock) can hinder parallelism. Also, fine-grained locks may be automatically crafted, to some extent, by using recent techniques based on static analysis [Mannarswamy et al., 2010].

HyTM — Another mechanism we have presented is the combination of STMs in the fall-back path of HTM, also known as HyTMs. Its main advantage is to allow concurrent execution of hardware transactions and software ones, used in the fall-back. However, during their concurrent execution, both software and hardware transactions have to play along in order to preserve correctness. In our study we considered two state of the art HyTM proposals [Matveev and Shavit, 2013, Dalessandro et al., 2011], which we evaluated for the first time on a commodity HTM from Intel. These two HyTMs were enhanced with the proposal of Reduced Hardware Transactions that we have presented on Section 2.5.2.

3.3.2 Methodology and Testbed

The benchmarks used in this study were manually instrumented so that reads and writes inside atomic blocks invoke the STM-based synchronization (when it is being used). Naturally, this is not relevant for the case of pure HTM approaches, in which case such instrumentation is void. We shall additionally present results for automatic compiler-based instrumentation in the later Section 3.6.

All the benchmarks and synchronization techniques are implemented in C or C++. Each experiment that we conduct is the average of 20 executions. We use the geometric mean whenever we show an average of normalized results. We often show speedup results, which are relative to the performance of sequential, non-instrumented executions, unless stated otherwise. The reported measurements of power consumption were obtained via the Intel RAPL [David et al., 2010] facility and are restricted to the processor and memory subsystems. Recent studies [Hackenberg et al., 2013, Hähnel et al., 2012] show that the model used by Intel RAPL estimates quite accurately the power consumption, when compared to a power meter attached to the machine.

We used a machine equipped with the first generation of Intel processors that were released with HTM support, whose details are shown in Table 3.2. This choice was dictated by the requirement of using a processor equipped with HTM, which was limited to 4 cores (and 8 hyper-threads) at the time at which this study was conducted.

Table 3.2: Characteristics of the Haswell machine, with HTM support and 8 cores, which was used in the comparative study.

Resource	Description
Processor	Xeon E3-1275 v3 3.5GHz
Cores	4 (each with hyper-threading)
L1 Cache	32KB 8-way (per core)
L2 Cache	256KB 8-way (per core)
L3 Cache	8MB (shared)
Cache Line	64B
RAM Size	32GB
Operating System	Ubuntu 12.04

Application threads are bound to physical cores in a round-robin fashion: e.g., 4 threads are allocated uniformly, one per core. As a result, hyper-threading is only used when 5 or more threads are used. We used GCC 4.8.2 with all compiler optimizations enabled and Ubuntu 12.04.

3.3.3 Understanding and Tuning Intel RTM

Before comparing the considered synchronization mechanisms, we conduct a set of preliminary experiments to understand the limitations of Intel RTM due to its best-effort nature. We also seek to assess several alternative configurations of the coupling between HTM and its fall-back. That allows to demonstrate that there is a significant impact on the varying performance and efficiency of different fall-back configurations. The settings identified during these preliminary experiments will be adopted in the remainder of the study to ensure that the comparison is performed using an appropriately tuned HTM.

We begin in Section 3.3.3.1 by quantifying the limits and causes of aborts for Intel RTM. Then, in Section 3.3.3.2, we compare the performance and energy efficiency when using six locks implementations to implement the single-global fall-back mechanism of the HTM. This shall allow us to narrow down the multitude of combinations of the HTM and lock implementations assessed in our study. Next, in Section 3.3.3.3, we optimize RTM-GL with a recently proposed technique [Afek et al., 2013] aimed at reducing spurious hardware aborts. Lastly, we investigate when it is best to give up on hardware and trigger the fall-back path, in Section 3.3.3.4. As we shall see, in fact this last optimization requires extensive trial and error testing, which motivated our work, presented later in Chapter 5, aimed at automating the tuning of this configuration.

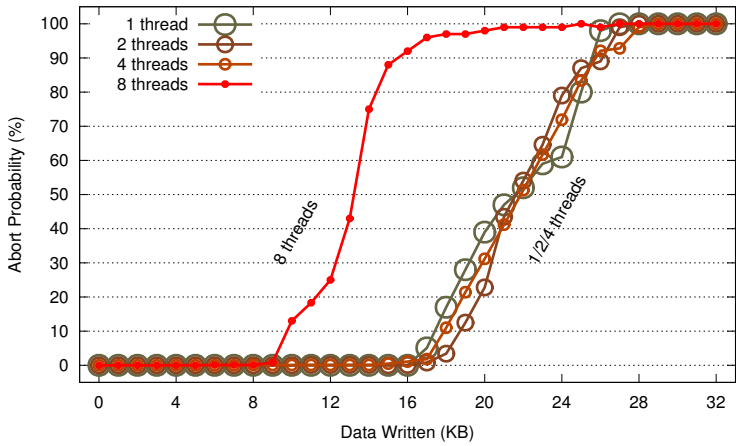


Figure 3.1: Probability that a hardware transaction aborts as a function of the amount of data **written**. We present this experiment when using a different number of threads.

3.3.3.1 Assessing the Limitations of Intel RTM

Although the hardware implementation details of Intel RTM are not formally available, black-box experimental analysis allows to understand some of its characteristics. In the following experiments, each data point corresponds to the average of 100 million transactions, to produce statistically meaningful results. Whenever we use more than one thread, we allocate each one to a different virtual core, and make sure we only use hyper-threads when all four physical cores are already taken (i.e., for more than 4 threads).

We begin by showing that writes are bounded by the L1 cache size, which in this processor has the size of 32KB per physical core. Figure 3.1 indicates the likelihood for a hardware transaction to abort when varying the amount of data written (over subsequent addresses at a granularity of the cache line size). We can see that no transaction is successful for a write set of the size of the L1 cache. Furthermore, the abort probability increases significantly as we write more than 20KB. Note also that the abort probability is only affected by the number of threads once we use hyper-threading (disabled up to 4 threads): this also strengthens the fact that writes are kept in the L1 cache, because it is shared only by co-located hyper-threads in the same physical core.

These behaviours were independent of the word size being written (up to 64 bytes), which also reinforces the idea that transactional accesses are tracked at the cache line granularity (possibly by marking a bit associated to the cache line as being written transactionally). As such, this

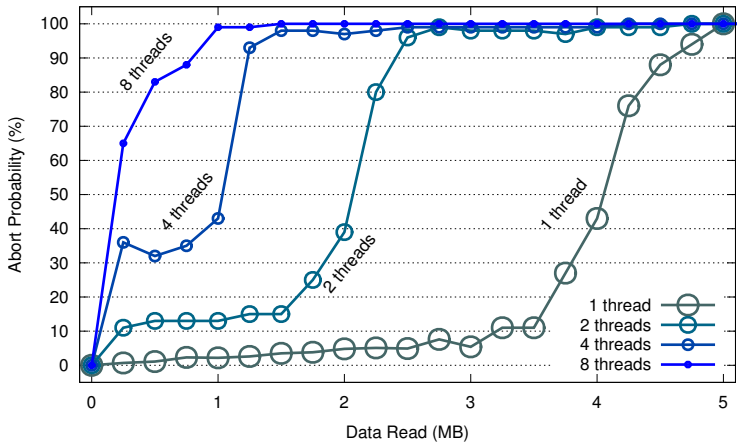


Figure 3.2: Probability that a hardware transaction aborts as a function of the amount of data **read**. We present this experiment when using a different number of threads.

also explains the non-negligible abort probability before exhausting the full L1 capacity: since the cache is 8-way associative, then more than 8 transactional writes that fall in the same cache line lead to an eviction from the L1 to the L2 cache. Since writes are tracked only in the L1, this also aborts the transaction.

We also study the limits associated to read accesses in Figure 3.2. The experiments reveal that it is possible to read far more than to write: a transaction, running alone in the system, was aborted consistently only for values larger than 5MB of data read. Hence, the read set tracking is definitely different from the way writes are tracked. This exploits the fact that the HTM need not recall the values that it read; it must only track the addresses, for instance with an over-approximation using a Bloom Filter. This is in contrast with the writes of a transaction, which must be recorded because at commit time we need to have exact information on both the value to be written and the address where to write it, in order to execute the write-back phase of the transaction and make the writes available for other threads.

We also show that the success of transactions, when reading, degrades gracefully with the number of threads. This leads us to believe that the read-set tracking may overflow beyond the L1 cache possibly into the L3 cache. Recalling the characteristics of this processor, described in Table 3.2, we can see that the L3 is the only cache that is shared among the cores. Because our experiments show an upper bound on the data that we may read, which changes with the number of threads and is independent of hyper-threading, we can then conclude that the reads

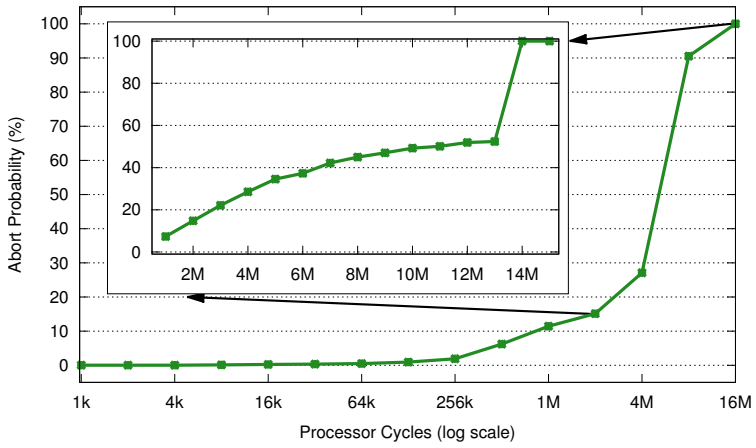


Figure 3.3: Probability that a hardware transaction aborts as a function of the length of a transaction in terms of processor cycles.

must be tracked in the only shared resource (the L3 cache). As a consequence of this design, this favours transactions that perform read operations earlier than write operations, because evicting cache lines read from the L1 may not abort the transaction (contrarily to the case of cache lines written).

Finally, we present Figure 3.3 where transactions run for an increasing amount of processor cycles, without performing any memory access. We highlight that, at roughly 14M cycles of length, all transactions abort. This corresponds to 4ms in our 3.5GHz processor, which is exactly the value set for our Linux kernel interrupt timer. As such, these transactions are aborted due to the periodic interrupt triggered by the kernel, which causes a control flow switch into privileged mode (typically, to run the kernel scheduler).

In fact, we have experimented also by modifying the kernel and allowing threads to run longer without getting interrupted, and verified experimentally that RTM transactions would run longer without getting aborted. We also verified that this is, in general in our benchmarks, not an issue because transactions are usually aborted for other reasons in long transactions (such as capacity overflows in the caches) long before they last enough time to get interrupted by the kernel.

To conclude this analysis, we summarize our findings with respect to Intel RTM in Table 3.3.

Table 3.3: Inferred characteristics of Intel RTM through black-box experimentation.

Category	Description
Conflicts	Memory accesses tracked at cache line size granularity.
Writes	Write-set tracking bounded by 32KB (the L1 cache size private to each physical core). Hyper-threads in same physical core contend for space to track writes.
Reads	Read-set tracking bounded by 5MB (order of magnitude of shared L3 cache space). All threads contend for space to track reads (when they are larger than 32KB, the L1 cache size).
Duration	The length of transactions is bounded in time by the Linux kernel periodic interrupt (default: 4ms).

3.3.3.2 The Impact of Locks on the Fall-back of HTM

The simpler way to use HTM is by relying on a single lock on the fall-back. Since the fall-back may be triggered often — by some thread, even if not all at the same time — the choice of which lock to use is quite important. Given the wide variety of lock proposals in the literature, we studied the six locks listed in Table 3.1 to be used in the fall-back. These implementations are representative of different design choices, and our goal is to understand if there is some implementation that consistently performs above the average across all parallelism degrees and benchmarks.

Table 3.4 shows the performance of Intel RTM given the backing lock implementation used in the fall-back path. We show the average overhead with respect to the best performing lock in each

Table 3.4: Overhead (%) of each lock implementation (as the fall-back of RTM) with respect to the optimal choice in each execution. We show both the overhead in terms of performance and power consumption, and in terms of cache misses in accessing memory.

	(Performance)		Cache	(Power)	
Lock	Overhead (%)	Rank	Misses (%)	Overhead (%)	Rank
Ticket	1.0	1.75	0.75	1.1	1.75
MCS	2.4	2.62	0.43	1.2	2.25
CLH	2.9	3.62	0.68	2.4	3.38
RW	14.2	4.89	5.71	17.4	3.88
TTAS	15.2	5.00	7.01	17.4	4.88
Spin	16.4	5.00	9.87	17.5	4.88

experiment (i.e., an idealized lock, because it is not necessarily the same across experiments), considering both time to complete the benchmark as well as power consumed. The reported overhead is the average across all STAMP benchmarks and thread counts (1 to 8). Using this metric, we can see that the Ticket, MCS and CLH locks perform best.

For each benchmark and thread count, we additionally sorted the considered lock implementations according to either their performance or power consumption, determining in this way their rank for that benchmark/configuration. This shows that no lock implementation is always the best or worse. However, we can see that the Ticket lock is consistently ranked higher, for which reason we shall rely on it from now on whenever we require locking (both standalone, or in the fall-back of RTM). Its performance is also tightly related with the overhead in terms of cache misses (measured with the *perf* tool in Linux). The Ticket lock indeed seems to strike a balance between being lightweight (in contrast to MCS and CLH) and avoiding repeated expensive synchronization operations (such as compare-and-swaps for the other implementations).

3.3.3.3 Improving the Single-lock Fall-back Path of HTM

The correct usage of the single-global lock fall-back requires hardware transactions to subscribe to the lock. As such, transactions that abort often and trigger the fall-back cause aborts of the hardware transactions, which can generate a chain effect, also known as *lemming effect* [Dice et al., 2012a], where the aborted hardware transactions also try to acquire the lock, preventing hardware speculation from ever resuming.

In [Afek et al., 2013], the authors use an auxiliary lock to prevent the lemming effect, while at the same time trying to preserve some concurrency. The idea is to guard the global lock acquisition by another lock. Aborted hardware transactions have to acquire this auxiliary lock before restarting speculation, which effectively serializes them. However, this auxiliary lock is not added to the read-set of hardware transactions, which avoids aborting concurrent hardware transactions. If this procedure is attempted some times before actually giving up and acquiring the global lock, then the chain reaction effect can be avoided: the auxiliary lock serves as a manager preventing hardware aborts from continuously acquiring the fall-back lock and preventing hardware speculations.

In Table 3.5 we compare Intel RTM using the auxiliary lock against the simple single-global lock (RTM-GL). For this, we report values for time, energy, and Energy Delay Product (EDP),

Table 3.5: Relative performance of RTM-GL over the auxiliary lock [Afek et al., 2013] across benchmarks and threads (higher values meaning that auxiliary lock took less time or consumed less energy).

Avg across Benchmarks				Avg across Threads			
benchmarks	time	energy	edp	threads	time	energy	edp
genome	1.58	1.6	2.54	1	1.00	1.00	1.01
intruder	1.80	1.95	3.52	2	1.08	1.06	1.14
kmeans	1.20	1.17	1.40	3	1.14	1.12	1.28
labyrinth	1.01	1.01	1.01	4	1.29	1.26	1.62
ssca2	1.00	1.00	1.00	5	1.26	1.25	1.57
vacation	1.52	1.48	2.25	6	1.26	1.23	1.55
yada	0.96	0.96	0.92	8	1.26	1.23	1.55

normalized with respect to RTM-GL (analogously to traditional speedup metrics). We report the average across either benchmarks or threads. Naturally, we can see that there is no difference with 1 thread because there is no concurrency and hence no problem resuming speculative execution. But beyond that, and in particular at higher concurrency levels, this technique helps consistently to improve the EDP. Some benchmarks do not show any difference because there are very little aborts (SSCA2) or RTM is not able to execute speculatively most of the time (Labyrinth). Yada's workload is conflict-intensive, for which reason the non-optimized approach is slightly better due to its inherent pessimism in following the fall-back path — that pays off since the high conflict probability limits the effectiveness of optimistic transactions.

3.3.3.4 Retry Policy for the Fall-back

Given that RTM must always have a fall-back due to its best-effort nature, an important decision is when to trigger that path. Upon a transaction abort, RTM provides an error code that informs about the reason of the abort. An abort due to a capacity exception is typically a good reason to trigger the fall-back path. However, hardware transactions may abort for various micro-architectural conditions that are less deterministically prone to happen upon transaction re-execution, and even capacity exceptions may not always be deterministic. Also, of course, transactions may abort due to data contention. In these situations one may aggressively trigger the fall-back, or opt to insist on using HTM.

As we will shall discuss in more detail in Section 3.6, the optimal choice of the retry policy can vary significantly across workloads and degrees of parallelism. As it is impractical to assume

Table 3.6: Summary of results according to the workload characterization of the STAMP suite. In some cases we denote different characterizations according to the number of threads used.

	Time in Tx (%)	Contention	Best Performing	Least Power Consumption
kmeans	low (7)	low	RTM-GL	RTM-GL
ssca2	low (17)	low	RTM-GL	RTM-GL
intruder	medium (33)	high	(RTM-GL $\leq$ 4threads $\wedge$ TinySTM $\geq$ 5 threads)	(RTM-GL $\leq$ 5threads $\wedge$ TinySTM $\geq$ 6threads)
vacation	high (89)	low	(RTM-GL $\leq$ 2threads $\wedge$ TinySTM $\geq$ 3threads)	(RTM-GL $\leq$ 4threads $\wedge$ TinySTM $\geq$ 5threads)
genome	high (97)	low	TinySTM	TinySTM
yada	high (99)	medium	SwissTM	TinySTM
labyrinth	high (100)	high	STMs (except TL2)	STMs (except TL2)

that the retry policy is ad-hoc tuned by programmers for each and single workload/application, we set the number of retries to 5, which is the configuration reported to deliver best all-around performance with RTM [Yoo et al., 2013, Karnagel et al., 2014] (a result that we have confirmed with RTM-GL on our testbed). For the HyTMs, 4 times was found to be the best number of retries on average.

3.4 Comparison in the STAMP Benchmark Suite

In this section we rely on the STAMP benchmark suite to assess the efficiency of all the synchronization mechanisms listed in Section 3.3.1, namely HTM, STMs, and HyTMs. We defer the comparison with fine-grained locking for Section 3.5.

We start by summarizing our results in Table 3.6. There, we list the STAMP benchmarks sorted by two important characteristics of their workloads: the contention level between transactions, and the percentage of the workload that is transactional. We then identify the TM mechanism that takes the least time to complete and which one consumes the least power, given the averaged results across threads.

This summarized perspective allows to highlight an interesting fact. It is possible to distinguish three categories in which RTM behaves differently, according to the transaction's characteristics. Kmeans and SSCA2 represent workloads with small transactions, medium frequency and low contention; here, RTM-GL performs consistently better than the alternatives across all threads. Intruder and Vacation exhibit medium profiles for what concerns the time spent in

transactions and contention; in these cases, RTM-GL results in the best performing solution using up to 4, respectively 2, threads, and the most energy efficient up to 5, respectively 4, threads. Finally, the other benchmarks spend almost all the time running transactions, encompassing both low and high contention scenarios. In these settings, TinySTM emerges as the most robust solution, both from the perspective of energy and performance.

This analysis allows to draw a set of guidelines to select which synchronization to use, at least when considering applications having analogous characteristics to those included in the STAMP suite. RTM-GL is desirable when transactions are small, generate low/medium contention, and the application does not spend all the time executing transactions. When contention increases, or the frequency of transactions is high, RTM-GL is competitive up to a medium degree of parallelism. In the remaining cases, STM is often the best choice, even when compared with fine-grained locking. The considered HyTMs perform poorly compared to the alternatives, never clearly outperforming the competing schemes in any benchmark. In the following Sections 3.4.1-3.4.2 we present our detailed experiments with STAMP.

3.4.1 Performance Study

In Figure 3.4 we show, for each benchmark and while varying the parallelism level, the speedup of all the considered synchronization schemes (with the exception of schemes based on fine-grained locking, which shall be presented in Section 3.5) with respect to a sequential, non-instrumented execution, and the power consumption during the execution (in Kilo Joules). This allows us to discuss in detail the differences between the mechanisms in different workloads.

Kmeans: This benchmark yields the biggest gap in performance between a RTM variant and STMs. Namely, RTM-GL reaches 3.5× speedup over a sequential execution, beating every other alternative both performance-wise and in terms of energy-efficiency. An interesting trend concerning energy-efficiency is that the power consumption with RTM (and, to some extent, also for all other synchronization schemes but GL) tends to slightly decrease as the parallelism level grows, which is a symptom of efficient utilization of the available architecture resources achievable using TM-based solutions. If we consider RTM-TL2 and RTM-NOrec, they are still competitive and better than the corresponding STMs, but they are far from RTM-GL in both metrics. It is worth noticing that the small and rare atomic blocks of this benchmark allow the GL approach to scale up to 3 threads. This explains the considerable success of RTM-GL in this

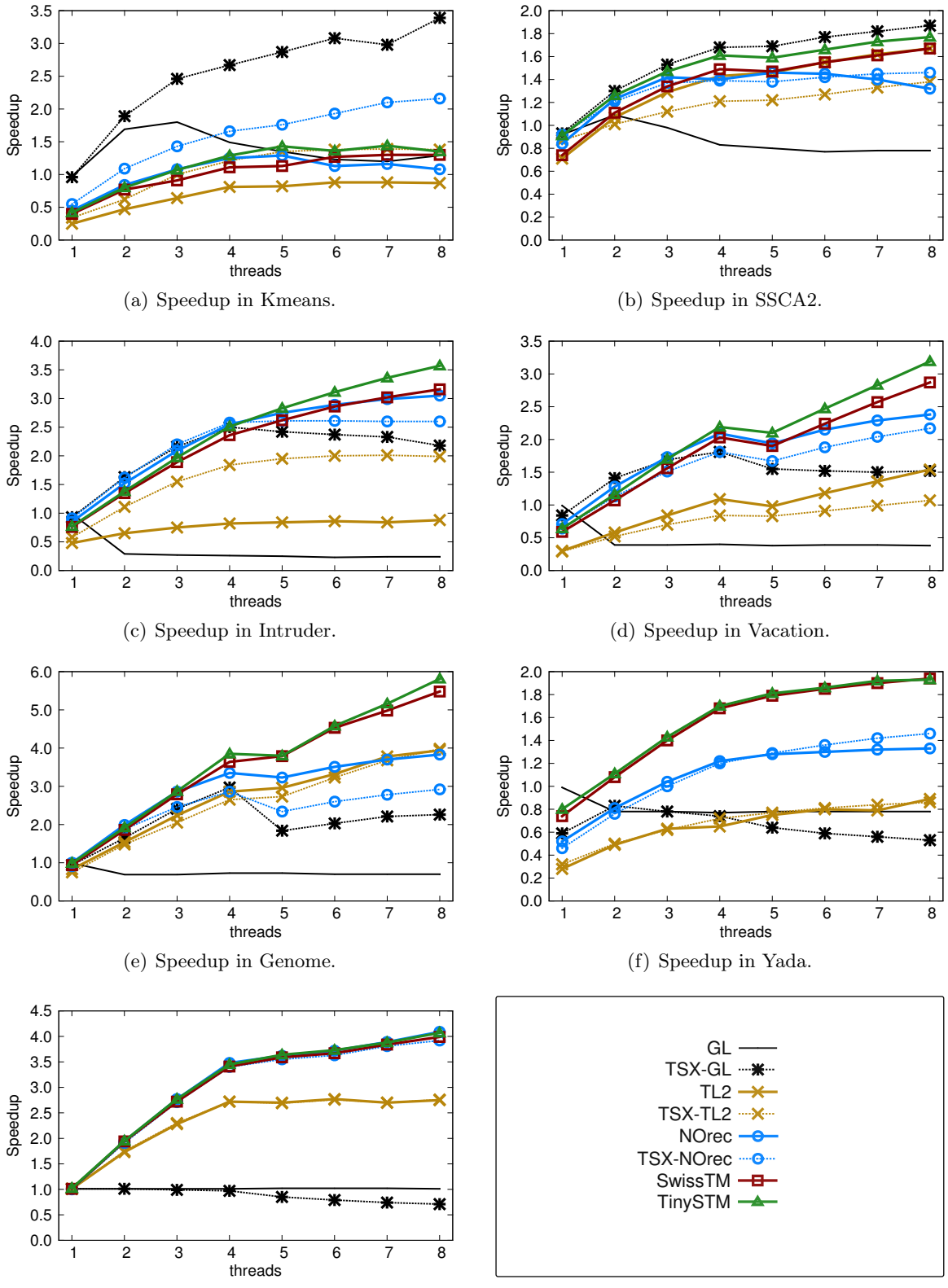
benchmark, as a transaction that resorts to the GL is still able to run concurrently with other threads that are not under an atomic block at that time.

SSCA2: This benchmark shows a similar trend between RTM variants, but with the significant difference that all STM approaches scale better as the degree of parallelism increases. Here, RTM-GL is only slightly better than the best STM, and this is consistent across all the thread counts. Also interestingly, RTM-TL2 improves little and fares rather bad on the energy side. This, however, is not the case for TL2 or RTM on their own, and as such is an artefact of the hybrid implementation integration. Finally, the reduced time within atomic blocks still allows the GL approach to scale up to 2 threads, which justifies the advantage of RTM-GL. However, this effect is smaller than in Kmeans, which also matches the fact that RTM-GL achieves less improvements over other approaches.

Intruder: Here RTM-NOrec, and RTM-GL to some extent, are competitive and even better (until 5 threads) than the best STMs (except for TL2). Since TL2 performs poorly in this benchmark, this also drags RTM-TL2 behind in both metrics. Interestingly, both TL2 and RTM-TL2 improve slightly performance with more threads, but TL2 consumes more power whereas RTM-TL2 slightly decreases it.

Vacation: Once again we see that the performance of RTM-TL2 is quite disappointing, as indeed TL2 itself performs poorly in this scenario. As we shall see throughout this study, TL2 is by far the worst STM among those considered, which is a result of having a similar algorithmic and synchronization complexity to that of SwissTM and TinySTM, while detecting conflicts lazily at commit-time. This results in TL2 doing useless work more often, whereas SwissTM and TinySTM restart the speculation faster when reacting to conflicts. On the other hand, NOrec is simpler, both in algorithmic as well as synchronization terms, reducing its instrumentation overheads and maximizing its performance at low thread counts. With regard to the other approaches, RTM-GL and RTM-NOrec are competitive with STMs until 4 threads. At higher parallelism degrees, their performance degrades due to contention on L1 caches caused by hyper-threading. Analogous results are achieved for what regards power consumption. It is interesting to note that RTM-GL performs worse than RTM-NOrec at 8 threads, but the two consume approximately the same power. This is a result of the power savings that are achievable with the lock acquisition in RTM-GL.

Genome: In the three last benchmarks we have either transaction-heavy or high-contention



(a) Speedup in Kmeans.

(b) Speedup in SSCA2.

(c) Speedup in Intruder.

(d) Speedup in Vacation.

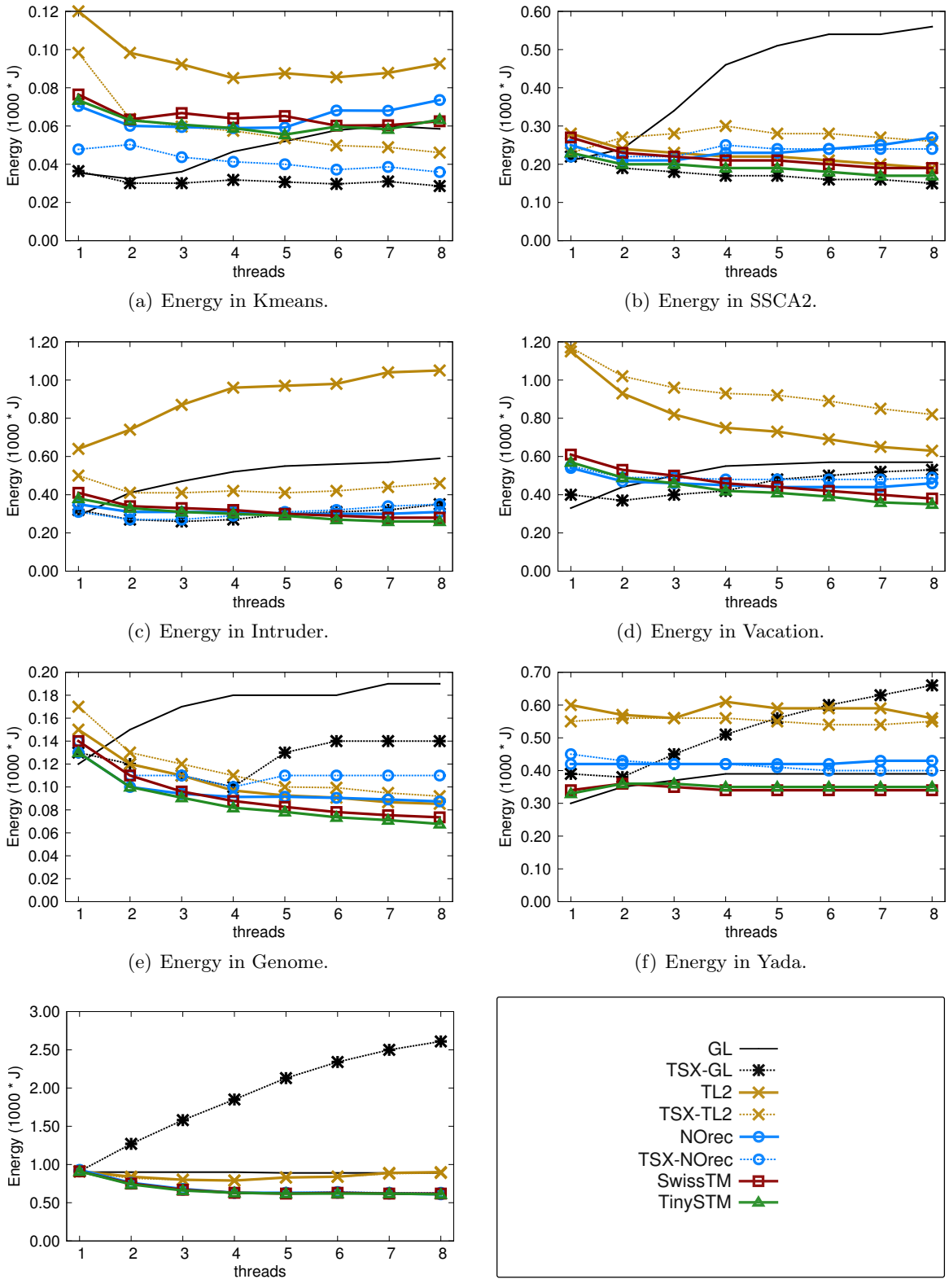
(e) Speedup in Genome.

(f) Speedup in Yada.

(g) Speedup in Labyrinth.

(h) Labels for the synchronization techniques.

Figure 3.4: Speedup (relative to non-instrumented sequential execution) when varying the number of threads in the STAMP suite.



(a) Energy in Kmeans.

(b) Energy in SSCA2.

(c) Energy in Intruder.

(d) Energy in Vacation.

(e) Energy in Genome.

(f) Energy in Yada.

(g) Energy in Labyrinth.

(h) Labels for the synchronization techniques.

Figure 3.5: Energy Consumption (in Kilo Joules) when varying the number of threads in the STAMP suite.

workloads, characterized by large transaction foot-prints. These conditions are clearly a much more favourable playground for STMs. In this case, we see a clear (and consistent across benchmarks) distinction between TL2 and NOrec, as these two lag behind in both metrics particularly at higher thread counts. Interestingly, we can see that RTM-TL2 performs best among the RTM variants at a higher concurrency degree, which is a singularity among all benchmarks. This benchmark also shows a clear trend when the 5^{th} thread is used: all approaches stabilize (or even decrease) performance at that point, due to hyper-threading. Interestingly, this effect is not so noticeable on the energy side, as STM approaches are still able to reduce the power consumed as parallelism increases. This highlights an interesting trade-off of hyper-threading: it allows sub-linear speed-ups only, but it also consumes little additional power. This fact is favourable to STMs, as RTM approaches generate more transactional aborts when hyper-threading is used, due the higher contention on the L1 cache.

Yada: This benchmark shows one scenario where RTM-GL performs poorly, with slowdowns above 3 threads. HyTMs follow closely their fall-back STMs' performance, as RTM is not able to succeed. This is also a case where TinySTM and SwissTM perform better than the other two STMs. This benchmark presents no surprises in the energy-efficiency, whose trends are highly correlated with the performance.

Labyrinth: Here we see STMs performing best and very alike each other. RTM-GL does not improve with thread count, simply because most transactions exceed the hardware cache capacity and, as such, eventually follow the fall-back path which is a sequential bottleneck given the GL. For this reason, RTM-TL2 and RTM-NOrec obtain some improvements, exactly because the fall-back allows for concurrency, contrarily to the global lock on RTM-GL. This scenario highlights, however, that HyTMs are capped by either RTM or the fall-back STM — as such, it is dubious whether they are practical (at least when used with RTM), or if it would be preferable to adaptively employ the most promising technique (RTM-GL or an STM) based on the workload.

3.4.2 Insights on TM Efficiency

In this section we shed some additional light on the factors dictating the trends observed in the experiments above. To this end, in Table 3.7, we report the average abort rate across benchmarks and threads for each synchronization mechanism. This represents the percentage of transactions that do not complete. Since there are four STMs under evaluation, we show the

minimum and maximum abort rates among them — typically the smallest abort rate belongs to TinySTM and SwissTM, whereas TL2 yields the maximum abort rate.

Once again, we structure the table considering the different categories of workloads. As we move right (more contended or transaction-intensive workloads) and down (higher degree of parallelism), RTM-based mechanisms increase the abort rates, which causes the loss of efficiency shown in the previous section. These results highlight that RTM has non-negligible aborts in many occasions where STMs abort very little.

In Figure 3.6 we consider four different benchmarks, representative of scenarios that allow to derive insights on the efficiency of the considered RTM variants. In those plots we present a breakdown of the reasons motivating transactional aborts, for each RTM mechanism. We distinguish aborts caused by exceeding the capacity of the processor's caches (as *capacity*); micro-architectural instructions or states forbidden by RTM, such as some system calls (as *architectural*); data contention resulting in conflicts (as *conflict*); and interaction between RTM and the fall-back paths, such as checking if the GL is free in RTM-GL or more complex logic in the case of HyTMs (as *interaction*).

Kmeans' breakdown shows that, as expected, as concurrency increases, also abort rates increase due mainly to data conflicts. It is worth mentioning that Kmeans is the benchmark with the least average aborts for RTM variants. Half of the aborts are due to conflicts, whereas the rest is motivated by a non-negligible percentage of aborts due to architectural instructions. This is something intrinsic to RTM, which is common throughout different benchmarks. The fact that these aborts occur less often in this benchmark allows RTM to obtain the most favourable results among all benchmarks.

In Yada and Labyrinth, instead, the workloads are much more transaction-intensive with non-negligible conflict rates. On top of this, the capacity of the caches is often exceeded by the hardware transactions (this is particularly visible in Labyrinth, where this phenomena dominates the aborts). This explains why the RTM variants followed up closely the performance of their fall-backs (with some constant overhead). HyTMs have a reduced abort rate because the fall-back's software transactions are also taken into account in these statistics, on top of the hardware transactions — since software transactions have little aborts due to the uncontended workload, they amortize the overall abort rate. In RTM-GL, instead, the fall-back executes non-speculatively due to the global lock, so we only count statistics for the hardware transactions

Table 3.7: Transactional abort rate (%). For STM we show the lowest and highest abort rate values obtained (across all considered STMs).

	benchmark	*kmeans*	*ssca2*	*intruder*	*vacation*	*genome*	*yada*	*labyrinth*
1 thread	STM	0 - 0	0 - 0	0 - 0	0 - 0	0 - 0	0 - 0	0 - 0
	RTM-GL	0	0	7	49	11	46	95
	RTM-TL2	0	0	36	94	35	19	53
	RTM-NOrec	0	0	4	40	6	19	53
4 threads	STM	10 - 34	0 - 0	0 - 0	0 - 6	1 - 51	5 - 58	4 - 13
	RTM-GL	26	0	22	69	31	48	100
	RTM-TL2	50	74	74	100	45	84	60
	RTM-NOrec	31	46	29	66	17	31	55
8 threads	STM	25 - 54	0 - 0	3 - 57	0 - 10	0 - 1	7 - 65	8 - 23
	RTM-GL	42	1	33	72	48	47	100
	RTM-TL2	60	99	92	100	53	92	69
	RTM-NOrec	44	88	62	99	69	39	60

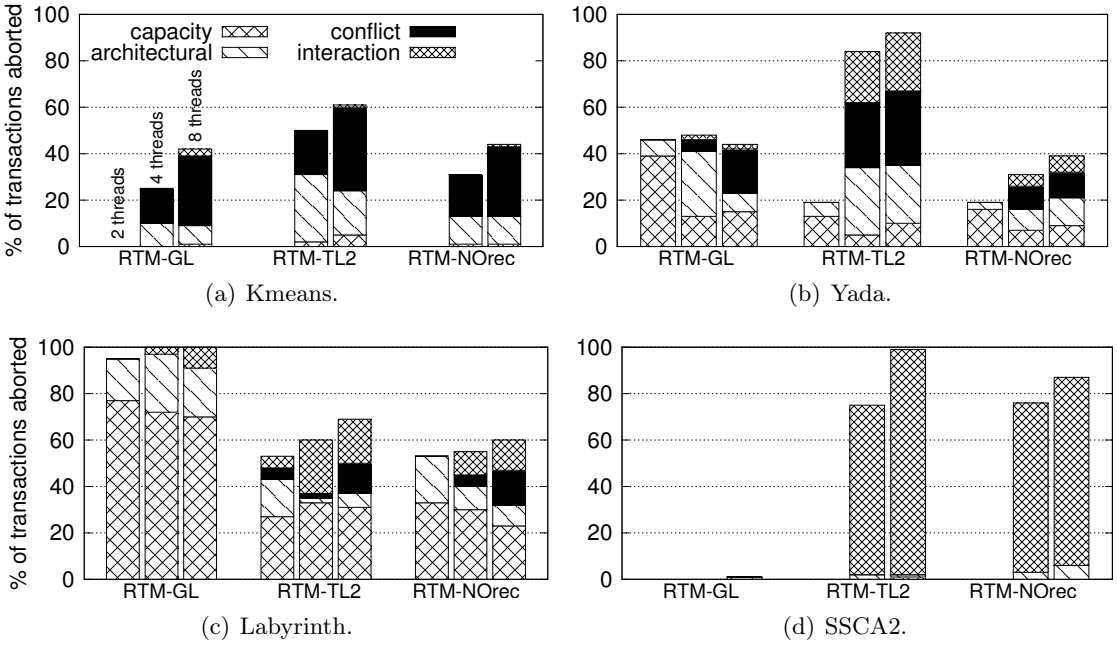
there.

Finally, SSCA2 shows a completely different scenario, in which RTM-GL generates almost no aborts (in line with STMs' behaviour), whereas HyTMs have enormous abort rates, dominated by the interaction with the fall-back path.

This motivates to better understand the usage of the fall-back path in the HyTMs. Table 3.8 shows the percentage of transactions that were executed in the fall-back (i.e., not purely in hardware). We show also, for those that triggered the fall-back, which percentage were able to execute in a fast mode, i.e., a mode in which the transaction executes in software but whose commit is boosted by using a reduced hardware transaction [Matveev and Shavit, 2013] (as explained in Section 2.5.2). For every table cell we show the percentage corresponding to 1 and 8 threads. Overall, the percentages vary linearly from 1 to 8 threads, for which reason we omit the intermediate values.

We start by highlighting in SSCA2 how both HyTMs are able to execute purely in hardware with 1 thread (they trigger the fall-back $< 1\%$ of the transactions). However, a higher thread count typically results in executing in the fall-back mode almost all the time, which matches the idea conveyed by Figure 3.6(d). In particular, for this benchmark, the ability to rely on hardware to speed up the software fall-back path is reduced from above 90% to 14% or even less.

These results for HyTMs show that RTM-TL2 triggers the fall-back more often, and is able to execute in the fast mode less frequently than RTM-NOrec. This justifies the advantage of



(a) Kmeans. (b) Yada.

(c) Labyrinth. (d) SSCA2.

Figure 3.6: Ratio of aborted transactions, among all attempted, identified also by the type of the abort for all RTM variants.

RTM-NOrec, which fared better across all benchmarks in Section 3.4: RTM-NOrec executes the fall-back software transactions in fast mode for most of the time. The reason is that the much simpler design of NOrec allows for a much easier integration with RTM in a HyTM.

Ideally one may want to also rely on more scalable STMs, like TinySTM or SwissTM, in the fall-back of RTM. However, due to the higher complexity of their algorithms, coupling them efficiently with HTM is a challenging task, and, in fact, we are not aware of any proposal in this sense in literature.

Finally, it has been pointed out in [Dalessandro et al., 2011] that, in order to support efficient HyTMs, it is desirable to have hardware support for selective non-transactional memory accesses in the scope of transactions. Such a feature is not currently supported in RTM, whereas its inclusion was planned for AMD's HTM proposal [Christie et al., 2010] (which was never commercialized). Hence, an interesting research direction suggested by this study is to investigate the impact of supporting non-transactional accesses, not only in terms of performance and energy, but also in terms of architectural intrusiveness.

Table 3.8: Ratio (%) of triggering the fall-back on HyTMs. We also show, of those that triggered the fall-back, which ratio executed *fast*, i.e., by using reduced hardware transactions. Intervals of values are shown, ranging from 1 (lower) to 8 threads (upper bound).

	RTM-TL2		RTM-NOrec	
	fall-back	Fast	fall-back	Fast
kmeans	< 1 - 77	92 - 32	< 1 - 78	100 - 23
ssca2	< 1 - 99	91 - 2	< 1 - 86	95 - 14
intruder	33 - 88	98 - 39	3 - 55	100 - 52
vacation	94 - 100	43 - 3	38 - 99	100 - 89
genome	50 - 100	97 - 71	6 - 67	100 - 94
yada	18 - 78	50 - 34	17 - 32	99 - 82
labyrinth	58 - 100	14 - 2	54 - 98	10 - 3

3.5 Benchmarks Using Fine-grained Locking

Most of the STAMP benchmarks have an irregular nature, which makes it very challenging to derive fine-grained locking schemes. In this section we focus on benchmarks for which it is possible to use (possibly very complex) fine-grained locking approaches. We start, in Section 3.5.1, by focusing on a subset of three STAMP benchmarks, for which we could craft an ad-hoc fine grained locking strategy. We then present results for Memcached in Section 3.5.2 and for two concurrent data structures in Section 3.5.3.

3.5.1 Fine-grained Locking in STAMP

As already mentioned, implementing a fine-grained locking strategy is a complex task for most of the STAMP benchmarks. We were still able to devise fine-grained locking strategies for three of the STAMP benchmarks, whose results we report in Figure 3.7. Besides fine-grained locks (FL), we also show results for RTM-FL, which combines hardware transactions with a fall-back path that relies on FL. Naturally, the combination of both schemes in RTM-FL requires hardware transactions to read (i.e., subscribe) all necessary locks as being free. We then compare these two approaches with RTM-GL and TinySTM, which were the best mechanisms in our previous experiments, and remove the others to improve the readability of the plots.

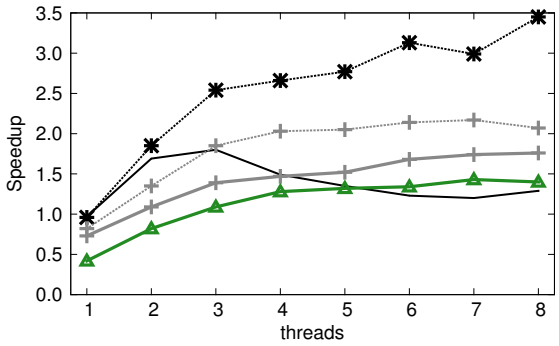
RTM-FL presents one advantage over RTM-GL, in that the fall-back path allows for threads to proceed in parallel if they require different locks (which is highly likely if there is little data

contention). However, this has the drawback that more locks have to be checked (during speculative executions) or acquired (during the fall-back executions). Hence, there is a clear trade-off that is subtle and difficult to manage.

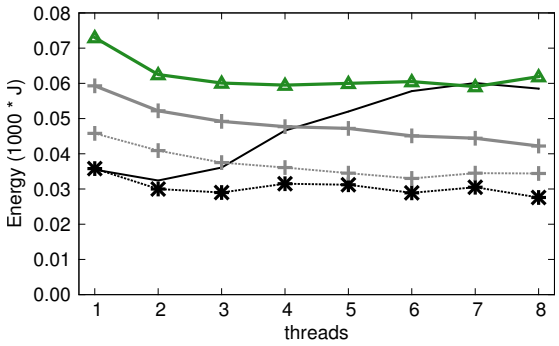
Recall that Kmeans and SSCA2 were the two benchmarks with workload characteristics more amenable to RTM-GL. This is justified by the low frequency of activation of the fall-back path. As such, RTM-GL incurs minimal overhead thanks to the hardware speculation and to the avoidance of any software-based instrumentation. Therefore, it is not a surprise that fine-grained locking is of no advantage in this scenario: each lock acquisition represents a synchronization point, whereas for RTM-GL there exists only explicit synchronization at the hardware level when a transaction attempts to commit. Note, however, that FL is consistently better than the best STM (TinySTM). This fact is even more relevant from the energy perspective, where the gap between FL and TinySTM is larger. Since the RTM fall-back is not triggered often, then RTM-FL goes through the additional verifications over more locks that are useless most of the time (to ensure a correct integration of the fall-back with hardware transactions), which explains its lower performance in this kind of workload.

In SSCA2 we see a different behaviour as both RTM variants perform quite similarly. This is explained by the fact that the fine-grained scheme is not very efficient: its locks are relatively coarse, which induces unnecessary serialization. This has the side-effect of making RTM-FL competitive with RTM-GL, because both have a similar effort in checking the locks in the speculative executions to ensure correct integration with the fall-back. Notice how the FL scheme still performs better than GL, which is a consequence of the higher degrees of parallelism achievable by reducing lock granularity. This confirms an expectable trade-off concerning lock granularity: the more fine-grained, the best the fall-back performs; however this can have an impact on the performance of the speculative executions as we saw for Kmeans.

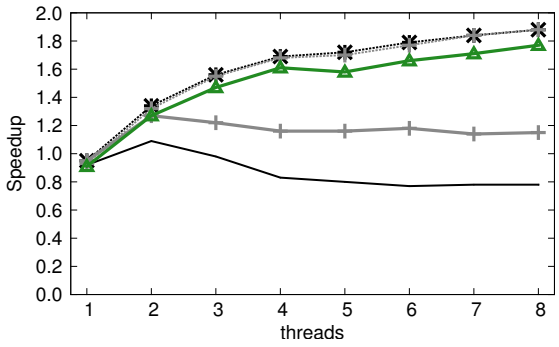
Finally, Intruder spends a large fraction of time within atomic blocks. As already discussed, this workload is more advantageous for STMs than for RTM. It is not surprising to see that RTM-GL is no longer the most competitive choice (although it still fares best until 3 threads). The interesting fact is that this kind of workload is more beneficial for FL. With more threads, TinySTM degrades its scalability, and is surpassed by FL. From an energy perspective, it is even clearer that FL is the best choice comparing to TinySTM, as it is almost always consuming less energy. RTM-FL suffers from the overheads of checking additional locks, until 3 threads, for



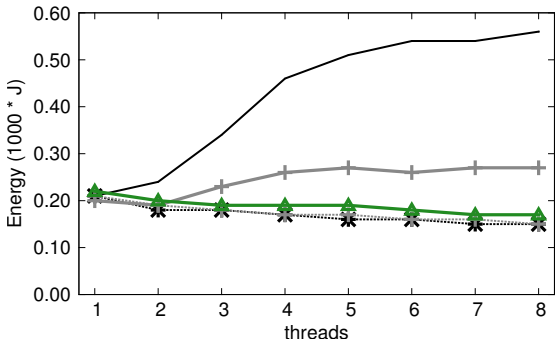
(a) Speedup in Kmeans.



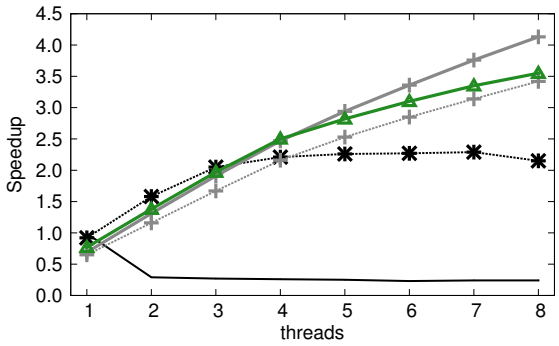
(b) Energy in Kmeans.



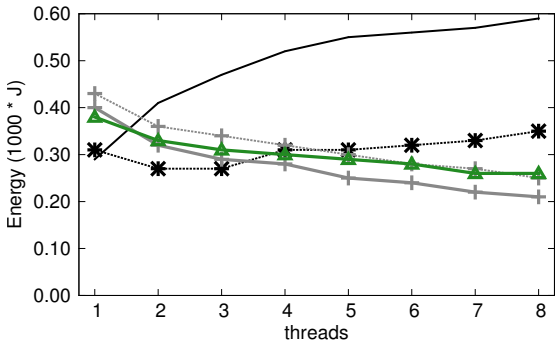
(c) Speedup in SSCA2.



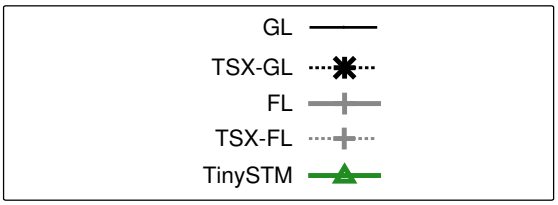
(d) Energy in SSCA2.



(e) Speedup in Intruder.



(f) Energy in Intruder.



(g) Labels for the synchronization techniques.

Strategy	10^3* Operations/sec
GL	223
RTM-GL	467
FL	501
RTM-FL	481
TinySTM	329

(h) Throughput in Memcached.

Figure 3.7: Experiment similar to that in Figure 3.4, using instead fine-grained locking, and showing also results in Memcached.

which reason it is not as good as RTM-GL. However, at that point RTM triggers the fall-back more often, which justifies the use of fine locks and allows RTM-FL to perform substantially better than RTM-GL. On the energy side, RTM-FL is closer to TinySTM, following the trend of STMs that often fare worse on the energy side to obtain comparative levels of performance to the other approaches.

3.5.2 Memcached

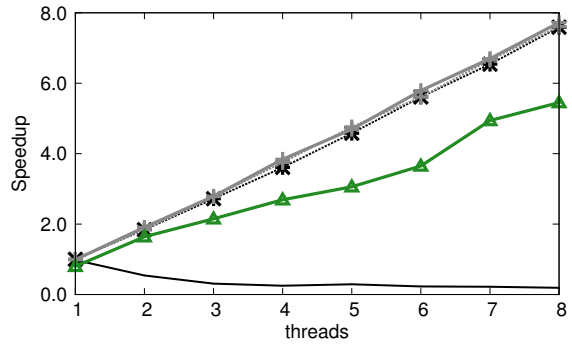
As presented earlier in Section 2.7, Memcached is a popular distributed object caching system. In this study, we rely on a recent TM-based porting [Spear et al., 2014], and use the original Memcached as the basis for FL. We used the *memslap* tool, configuring the workload with 95% lookups and 5% insertions, 8 threads and a concurrency of 256.

In Memcached it is not really possible to measure, for reference purposes, the performance of a sequential execution, because there is always concurrency due to the existence of a pool of maintenance threads. Hence, we present the peak throughput obtained using the maximum number of available hardware threads (see Figure 3.7(h)). The results show that FL has the best performance, but RTM-GL is only 7% behind. This is a significant achievement as the effort to devise such fine-grained locks is considerably higher than using RTM-GL. Also, since FL is quite optimized, it is expectable that RTM-FL is not able to extract any further parallelism. Interestingly, with this benchmark, TinySTM is not competitive because the instrumentation overheads are amplified by the short and uncontended transactions.

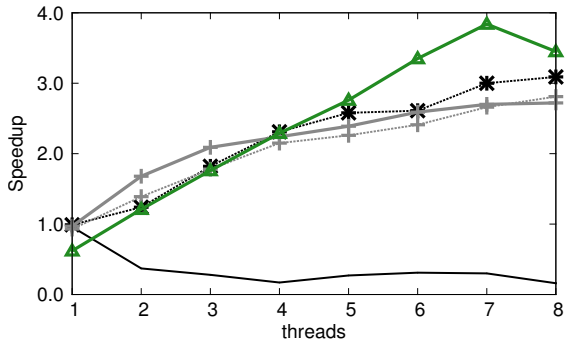
3.5.3 Concurrent Data Structures

We now consider two concurrent data-structures, namely a Red-Black Tree and a Hash-Map, which represent particularly relevant use cases for TM given the complexity of designing efficient fine-grained locking strategies for these scenarios.

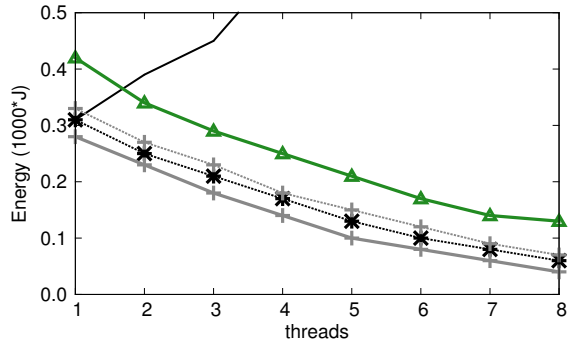
Figure 3.8 shows two different scenarios: we consider a small Hash-Map (512 buckets) with only 10% transactions performing writes (the rest are lookup operations), and a large Red-Black Tree (1 million items) with 90% transactions performing updates. In the former case, RTM-GL achieves perfect linear scalability, which is a consequence of its negligible overheads and of the



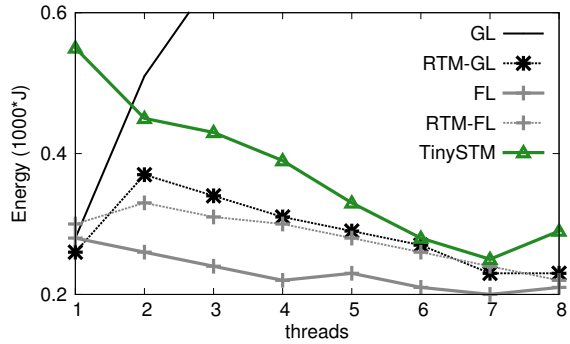
(a) Hash-Map 10% writes.



(b) Red-Black Tree 90% writes.



(c) Hash-Map 10% writes.



(d) Red-Black Tree 90% writes.

size and % of writes	10%	50%	90%
2^7 elements	3.30	1.79	1.21
2^{14} elements	2.96	1.58	1.11
2^{21} elements	1.86	1.33	1.06

(e) Normalized EDP of the best alternative to RTM-GL in Red-black Tree (higher is favourable to RTM-GL).

Figure 3.8: Data Structures varying contention level.

very reduced abort rate. With larger transactions, the gains achievable by RTM tend to diminish, although it still remains a very competitive solutions.

Table 3.8(e) shows a spectrum of workloads in Red-Black Tree, by considering the normalized EDP of RTM-GL against the best alternative in each experiment. For this, we vary the size of the tree and the percentage of write transactions. The trend is clear in this table: RTM behaves best with light workloads, and loses advantage when transactions become larger or write-intensive. This confirms the results of the analysis that we performed for STAMP, given that, also in this case, RTM shines most when atomic blocks have little duration and the workload is not fully transactional.

Table 3.9: Improvement of configuring RTM-GL for each workload compared to the single configuration used in our study.

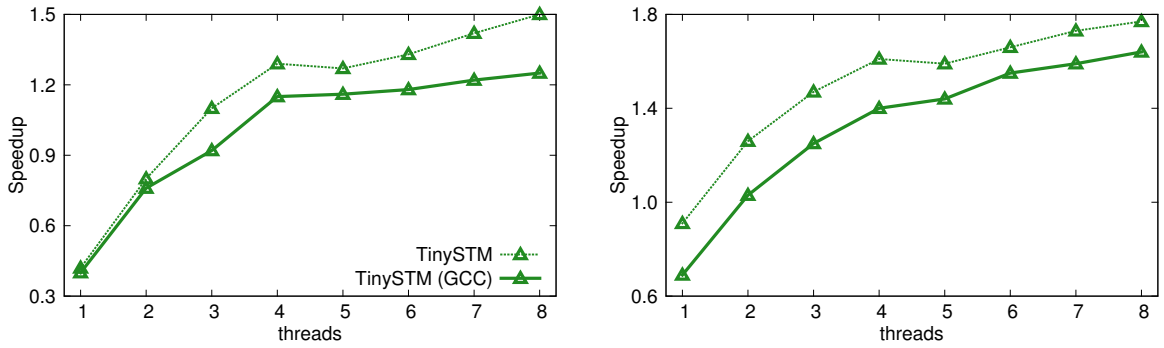
Speedup %	kmeans	ssca2	intruder	vacation	genome	yada	labyrinth
4 threads	12	7	20	36	12	13	2
8 threads	5	8	80	21	2	55	39

3.6 Research Directions Suggested by our Study

We now identify some relevant research directions that emerged from the analysis of our experimental study. In fact, some of the challenges mentioned intersect with those presented earlier in the summary of this dissertation, as we explored them further in the scope of this work.

• The overall performance of the tested HyTM solutions is quite disappointing. These findings contradict the simulation results published in several previous works, e.g., [Matveev and Shavit, 2013]. Our analysis suggest that the root cause of the problem is related to the inefficiency of the mechanisms used to couple hardware and software transactions, which is generating a large number of spurious aborts. However, further research is due in order to understand what can be done to address such a problem. An interesting research question, in this sense, is whether the availability of support for enabling non-transactional memory accesses while executing hardware-assisted atomic blocks could indeed allow for more efficient interplay between HTM and STM (which has been assumed by other works in the area of HyTM, e.g. [Riegel et al., 2011]). A related research question is how to support such a feature while minimizing the disruptiveness of the changes required at the hardware level — an aspect that cannot be overlooked given the complexity of modern processor architectures.

• As mentioned in Section 3.3.3, the performance of RTM is significantly affected by the retry policy (e.g., the settings of the number of retries upon abort, and the choice of how to react to capacity aborts). While in our study we used the configuration that performed best on average, as shown in Table 3.9, significant speedups (up to 80%) with regard to the configuration used in our study can be achieved by ad-hoc tuning the retry policy for the specific workload — even more could be achieved by considering the specific concurrency degree as well. Unfortunately, this is a tedious and error prone task that is not desirable to delegate to programmers. Hence,



(a) Kmeans. (b) SSCA2.

Figure 3.9: Impact of compiler instrumentation with GCC.

these findings highlight the relevance of devising solutions for adaptively tuning these parameters in an automated manner. The key challenge is how to do it with minimal overhead, given that the cost imposed by self-tuning approaches targeting STMs (based on complex machine-learning [Rughetti et al., 2012] or analytical models [Di Sanzo et al., 2012]) is going to be strongly amplified in HTM settings because there exists no instrumentation as in STMs. We pursue this further in Chapter 5.

• Our study has used selective (i.e., manual) instrumentation when considering both STMs and HyTMs, i.e. only the relevant subset of memory locations accessed in atomic blocks have been traced. As an alternative, one could rely on the compiler to automatically instrument atomic blocks with calls to the TM runtime. The plots in Figure 3.9, which were obtained using the C++ TM extension integrated in GCC 4.8.2, show that non-selective instrumentations can impact performance by approximately 20% when using TinySTM. This is a consequence of the increase of the transaction footprint (up to 3x larger with SSCA2) caused by the "blind" instrumentation performed by GCC.

Not only do these results unveil the possibility of optimizations in existing compiler's support for STM, but also provide an additional compelling motivation to incorporate support for selective instrumentation in HTM. Indeed, we have shown that capacity exceptions are one of the key sources of aborts with HTM. Hence, techniques capable of achieving noticeable reductions of the transactions' footprint are expected to strongly benefit HTM's performance. These considerations open interesting research avenues investigating cross-layer mechanisms operating at the compiler and architectural level, and aimed at supporting selective instrumentation in a way that is both convenient for the programmer (i.e., possibly fully transparent) and sufficiently

non-intrusive to simplify integration in existing architectures.

3.7 Summary

In this chapter we analyzed extensively the performance and energy efficiency of several state of the art TM systems. We compared different TM solutions (software, hardware and combinations thereof) among each other and against lock based systems. Our study demonstrates that the recent HTM implementation by Intel can strongly outperform any other synchronization alternative in workloads with small transactional foot-prints (and sometimes in conjunction with medium to low execution time spent transactionally). On the other hand, it also identified some critical limitations of Intel RTM, and highlighted the robustness of state of the art STMs. These software implementations achieve performance competitive with fine-grained locking, and outperform HTM in workloads encompassing long and contention-prone transactions. However, we have also confirmed that different TMs tend to out-perform each other across different workloads and applications.

Furthermore we have shown that the performance of HyTMs, when used in combination with RTM, is normally quite disappointing; we determined that the root cause of this surprising result lies in the inefficiency of the mechanisms used to couple software and hardware transactions. This means that it is better to choose between either pure HTM or STM for each workload. However, imposing that choice on the programmer is clearly a burden, which would be preferably avoided — this is an issue that we revisit in Chapter 7. Finally, our study allowed to identify a set of compelling research questions, which, we believe, should be timely addressed to increase the chances of turning HTM into a mainstream paradigm for parallel programming.

4 Time-Warp: Reduction of Conflicts in TM

As we have seen so far, mainly on the background provided on TM algorithms in Chapter 2, TMs allow concurrent executions to remain correct by tracking which memory locations are accessed transactionally. This information is then used to detect conflicts by the TM runtime, and possibly abort transactions with the objective of guaranteeing a safe execution.

In the case of HTMs, this tracking is inherently coupled with the cache coherence protocol, which strongly limits the flexibility of implementing arbitrary concurrency control schemes (that is, to minimize the cost of changes in the processor's logic and the subsequent verification procedures [Adir et al., 2014]).

As for STMs, instead, given their pure software nature, the access tracking and conflict detection mechanisms can clearly be implemented in a much more flexible way. In fact, it is quite often the case that STMs are designed to minimize the instrumentation overhead by relying on simple concurrency control mechanisms — which are amenable of efficient implementations — but that are also prone to suffer of *spurious aborts*, i.e., they abort transactions unnecessarily, even when they do not threaten correctness.

Indeed, existing literature on STMs has highlighted an inherent trade-off between the efficiency of a TM algorithm, and the number of spurious aborts it produces — that is the notion of Permissiveness [Guerraoui et al., 2008], which we presented in Section 2.2.2. Recalling, a TM is permissive if it aborts a transaction only when the resulting history (without the abort) does not respect some target correctness criterion (e.g., Serializability or Opacity).

Achieving permissiveness, however, comes at a non-negligible cost, both theoretically [Keidar and Perelman, 2009] and in practice [Gramoli et al., 2010]. Indeed, most state of the art TMs, such as those studied in the previous chapter [Dice et al., 2006, Felber et al., 2008, Fernandes and Cachopo, 2011, Dalessandro et al., 2010], are far from being permissive. They tend to resort to concurrency control algorithms that generate a large number of spurious aborts, but which have the advantage of allowing highly efficient implementations.

4.1 The Problem

To illustrate the problem incurred by those STMs, consider an example consisting of a sorted linked-list as shown in Figure 4.1. This list is accessed by update transactions that insert or remove an element, and by read-only transactions that try to find out if a given element is in the list. Let us consider three transactions: a read-only transaction T_1 that seeks element D in the list; an update transaction T_2 that inserts item B; and an update transaction T_3 that removes item E. In the figure we also show a possible execution for the operations of each transaction, and the corresponding result, in a typical STM.

One widely used form of reducing spurious aborts is by serializing a read-only transaction R before any concurrent update transaction. The intuition is that read-only transactions do not write to shared variables and, consequently, are not visible to other transactions. Thus T_1 is allowed to commit in the example — many STMs skip validation for read-only transactions at commit-time [Dice et al., 2006, Felber et al., 2008, Fernandes and Cachopo, 2011] because they can be safely serialized in the past. Hence, in the example, we obtain the serialization order of T_1 preceding T_2, even though they execute the commit procedure in the opposite order.

Let us now consider T_3, which is an update transaction that modifies shared variables. The execution shown for T_3 dictates its abort in state of the art, practical STM algorithms, e.g., [Felber et al., 2010a, Dalessandro et al., 2010]. To minimize overheads, these STMs often rely on a simple validation scheme, which allows update transactions to commit only if they can be serialized at the present time, i.e., after every other transaction committed so far (in the example, time 6). This validation mechanism has been systematically adopted by a number of STM algorithms (and database concurrency control schemes [Bernstein et al., 1987, Adya, 1999]), for which reason we refer it as *classic validation* rule.

According to this rule, a transaction T aborts when conducting the validation (possibly at commit-time) if a concurrent transaction committed a new value to some variable, for which T read a stale value. In the example, when T_3 is validated at commit, the *next* pointer of element A is found to have been updated after T_3 read it, causing T_3 to abort. Notice, however, that this abort is spurious, given that T_3 could have been safely serialized "in the past", namely before T_2, yielding the equivalent sequential history $T_1 \rightarrow T_3 \rightarrow T_2$.

On the other hand, serializing update transactions in the past is not always possible, as

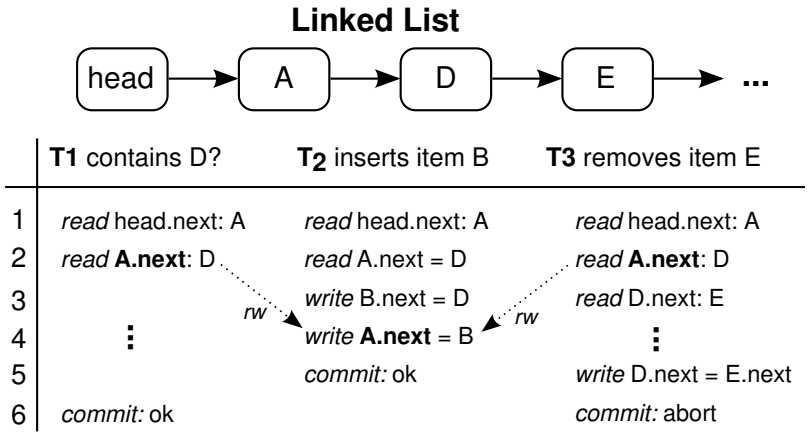


Figure 4.1: Possible execution for three transactions with an STM when accessing a sorted list.

their effects could have been missed by concurrently committed update transactions. This would be the case, for instance, if T_3 had also attempted to insert element C, missing the concurrent update of T_2 and overwriting A.next (de facto removing B from the list). If one considers the serialization graph [Adya, 1999], then in such a scenario the graph would be cyclic, for which the serializability theorem dictates that T_3 could not be spared from aborting [Bernstein et al., 1987]. The intuition is that it would be impossible to commit T_3 without creating a contradiction of dependencies between some committed transactions. We formally present these concepts, namely that of the serializability graph, later in Section 4.4.1.

Overall, minimizing spurious aborts, in a practical way, requires designing algorithms capable of deciding *efficiently* (i.e., without checking the full serialization graph) when update transactions can be serialized in the past.

4.2 Overview

In this chapter we present an algorithm to efficiently tackle the problem identified above: the Time-Warp Multi-version (TWM) is a multi-versioned STM that strikes a new balance between permissiveness and efficiency to reduce spurious aborts.

The key idea at the basis of TWM is to allow an update transaction, which missed some writes produced by a concurrent committed transaction T', to be serialized "in the past", namely before T'. Unlike TM algorithms that ensure permissiveness [Ramadan et al., 2009, Keidar and Perelman, 2009], TWM exclusively tracks the direct conflicts (more precisely, anti-dependencies [Adya,

1999]) developed by a committing transaction, avoiding onerous validation of the entire conflict graph. Thus, TWM's novel validation is sufficiently lightweight to ensure efficiency, but it can also accept far more histories than state of the art, efficient TM algorithms that only allow the commit of update transactions "in the present" (using the *classic validation* rule).

We also contribute to the theoretical knowledge on the reduction of spurious aborts by studying the Input Acceptance [Gramoli et al., 2010] of several STMs, including Time-Warp. Furthermore, with respect to progress guarantees, we provide a detailed algorithm to implement TWM with lock-freedom. Concerning safety properties, we prove that TWM ensures *Virtual World Consistency* (VWC) [Imbs and Raynal, 2012] (see Section 2.2.1), which we recall also provides consistency guarantees on the snapshots observed by transactions that abort (besides Serializability for committed transactions). This means that TWM prevents typical problems (such as infinite loops and run time exceptions) such as those that arise from observing inconsistent values, which would not be producible in *any* sequential execution.

We present an extensive experimental study comparing TWM with five other STMs representative of different designs, guarantees and algorithmic complexities. This study was conducted on a multi-core machine with 64 cores using a breadth of TM benchmarks. The results highlight gains up to 9×, with average gains across all benchmarks and compared TMs of 65% in high concurrency scenarios, thus contributing to improve the performance robustness in the scope of STMs.

Finally, we highlight also the generality and extensibility of this idea, namely by applying Time-Warping also to a Distributed TM (DTM) (such as those described in Section 2.6). We believe this is noteworthy, as it illustrates the significance of this contribution to several domains, hence increasing its relevance and widening its applicability.

The remainder of this chapter is structured as follows. In Section 4.3 we discuss related work. Then we provide a high-level overview of Time-Warp, and a simple lock-based implementation, in Section 4.4. Based on the rules that we present for Time-Warp, we then conduct a theoretical analysis on the correctness criteria achievable by TWM, as well as on its ability to avoid spurious aborts (Sections 4.5 and 4.6). Afterwards, we present an optimized algorithm for TWM in Section 4.7, with the main key feature of ensuring lock-freedom. Section 4.8 introduces our extensive experimental study. Finally we summarize the extension of Time-Warp to a DTM in Section 4.9.

4.3 Related Work

A well known technique to reduce conflicts is to design concurrency controls that ensure that read-only transactions can never be aborted. This idea has been formally characterized as MV-Permissiveness [Perelman et al., 2010], as explained in Section 2.2.2, which is typically implemented with multi-versioned STM algorithms [Perelman et al., 2011, Diegues and Cachopo, 2013, Lu and Scott, 2013] (although a single-version algorithm has been proposed also to provide it [Attiya and Hillel, 2011]). Here, we seek to reduce spurious aborts even further than MV-Permissiveness.

Several TM proposals were designed with the main concern of reducing spurious aborts. As presented earlier in the background on TM literature, there are works targeting different consistency criteria (Serializability, VWC, and Opacity) and pursuing Permissiveness using both probabilistic and deterministic techniques. These design decisions have a strong impact on several important details of these algorithms. Nevertheless, it is still possible to coarsely distinguish them into two classes:

1. Algorithms [Ramadan et al., 2009, Gramoli et al., 2010, Keidar and Perelman, 2009] that instantiate the full transactions' conflict graph and ensure consistency by ensuring its acyclicity [Papadimitriou, 1979];

2. Algorithms [Guerraoui et al., 2008, Aydonat and Abdelrahman, 2012, Crain et al., 2011] that determine the possible serialization points of transactions by using time intervals, whose bounds are dynamically adjusted based on the conflicts developed with other concurrent transactions (interval-based approaches).

Concerning the first class of algorithms, which rely on tracking the full conflict graph, these are generally recognized (often by the same authors [Gramoli et al., 2010, Keidar and Perelman, 2009]) to introduce excessive overhead to be used in practical systems. Interval-based algorithms, in the second class, have more efficient implementations but allow some spurious aborts. As such, they tend to have somewhat costly commit procedures that are not advantageous all the time, thus hindering their viability in various practical scenarios as we show in our evaluation.

In contrast, our Time-Warp proposal leverages on the lessons learnt from prior art and identifies a sweet spot between efficiency (i.e., avoiding costly bookkeeping operations) and the

ability to avoid spurious aborts: (1) TWM deterministically accepts many common patterns rejected by practical TM algorithms, by tracking only *direct* conflicts between transactions; and (2) it exploits multi-versioning to further reduce aborts and achieve MV-Permissiveness.

We summarize this comparison of Time-Warp, with other STMs that aim to reduce the number of spurious aborts, in Table 4.1.

We highlight that the Online Permissive STMs, which perform tracking of the full conflict graph, still yield some spurious aborts in practice (although less than other solutions) due to the need for performing deadlock/cycle detection over the conflict graph in a practical way. To do so, a practical implementation would typically resort to timeouts, which are thus responsible for some spurious aborts. In comparison, Time-Warping is also able to reduce spurious aborts with a decentralized approach — i.e., there is no single location for the whole meta-data of the system to be maintained — and by monitoring only direct conflicts between transactions — this is in contrast with the approaches that instantiate and maintain the whole graph of conflicts. Furthermore, it ensures a correctness criteria that avoids inconsistent snapshots for any transactions (as formally specified by the VWC criterion), and provides a strong progress guarantee.

Another approach that may serve for reduction of conflicts is Transactional Boosting [Herlihy and Koskinen, 2008]. Boosting creates a transactional object (namely, a collection) out of a black box concurrent implementation of that object. This is performed by using an input definition of the commutativity of each operation of the object specification (as well as inverse operations). As a result, conflict detection can be performed using higher-level semantics rather than low-level accesses to transactional variables. However, Boosting is not generalizable beyond objects whose operations commute. Furthermore, it is also a pessimistic approach that relies on locking, and can weaken the progress guarantees originally provided by the underlying STM.

TWM also shares commonalities with Serializable Snapshot Isolation (SSI) [Cahill et al., 2008], a technique proposed for Database Management Systems, which enhances Snapshot Isolation [Berenson et al., 1995] DBMSs to provide Serializability. In particular, both schemes track direct (anti-dependency [Adya, 1999]) conflicts between transactions to detect possible Serializability violations. However, the two algorithms differ significantly both from a theoretical and a pragmatic standpoint. First, unlike TWM, SSI does not ensure MV-Permissiveness (i.e., SSI can abort read-only transactions). Further, SSI was designed to be layered on top, and guarantee

Table 4.1: Comparison of STMs that aim to reduce the number of spurious aborts while preserving the TM abstraction unmodified and strong correctness criteria. In comparison to these, typical STMs such as TinySTM [Felber et al., 2008] and SwissTM [Dragojević et al., 2009a], tend to generate many spurious aborts.

STM Algorithm	Monitors Only Direct Conflicts	Decentralized	Permissiveness	Correctness Criteria	Progress Guarantee
DATM [Ramadan et al., 2009]	χ	χ	Online Permissive	Serializability	Deadlock-free
SSTM [Gramoli et al., 2010]	χ	✔	Online Permissive	Serializability	Deadlock-free
AbortsAvoider [Keidar and Perelman, 2009]	χ	χ	Online Permissive	Opacity	Deadlock-free
AVSTM [Guerraoui et al., 2008]	✔	✔	Probabilistic Permissive	Opacity	Lock-free
TSTM [Aydonat and Abdelrahman, 2012]	✔	✔	No	Serializability	Starvation-free
IR_VWC_P [Crain et al., 2011]	✔	✔	Probabilistic Permissive	VWC	Starvation-free
Time-Warp	✔	✔	MV-Permissive + TWM	VWC	Lock-free

inter-operability with, a Snapshot Isolation concurrency control mechanism designed to work in disk-based DBMS environments. Hence, SSI relies on techniques (e.g., a global lock-table that needs to be periodically garbage collected to avoid spurious aborts) that would have an unbearable overhead in a disk-less environment, such as in TM.

Finally, TWM draws inspiration from Jefferson's Virtual Time and Time-Warp concepts [Jefferson, 1985], which also aim at decoupling the real-time ordering of events from their actual serialization order. In Jefferson's work, however, Time-Warp is used to reconstruct a safe global state. In TWM, instead, the time-warp mechanism injects "back in time" the versions produced by transactions that observed an obsolete snapshot, with the ultimate goal of reducing spurious aborts.

4.4 Time-Warp

This section presents the Time-Warp Multi-version algorithm (TWM). We begin by introducing some preliminary notations and assumptions in Section 4.4.1. We then explain the idea of Time-Warp in high-level terms in Section 4.4.2. Finally, in Section 4.4.3, we present a simple lock-based implementation of our proposal. This allows us to focus on the core of the idea, avoiding introducing, at least for the moment, additional optimizations that would make it more complex to reason on its correctness. An optimized, lock-free version of TWM will be presented later, in Section 4.7.

4.4.1 Preliminary Notations and Assumptions

We consider a conventional transaction execution model [Adya, 1999, Bernstein et al., 1987] in which the transactions can generate the following set of operations: a transaction starts with a *begin* operation, followed by a sequence of *read* and *write* operations on shared variables (i.e., data items), and can be finalized either with a *commit* or *abort* operation. The operations of a transaction are totally ordered, and every transaction T_i is uniquely identified by their suffix i. Furthermore, two operations from different transactions are said to *conflict* if they operate on the same shared variable and at least one the operations is a write.

Given a set of transactions $\mathcal{S}$ produced by a run of a Multi-Versioning Concurrency Control (MVCC) TM algorithm, a history $\mathcal{H}$ over $\mathcal{S}$ is defined by two parts: $\prec_{\mathcal{H}}$ and $\ll$. The latter,

$\ll$, is a total order on the set of committed data item versions for each shared variable. The former, $\prec_{\mathcal{H}}$, is a partial order over the set of operations of all transactions in $\mathcal{S}$, such that: 1) $\prec_{\mathcal{H}}$ preserves the total order among each operation of $T_i \in \mathcal{S}$, and 2) every two conflicting operations are ordered by $\prec_{\mathcal{H}}$.

We denote with DSG($\mathcal{H}$) a Direct Serialization Graph over a history $\mathcal{H}$, i.e., a directed graph containing: a vertex for each committed transaction in $\mathcal{H}$; an edge from a vertex corresponding to a transaction T_i to a vertex corresponding to transaction T_j, if there exists a read/write/anti-dependency from T_i to T_j. These edges are labelled with the type of the dependency: (1) $A \xrightarrow{wr} B$ when B read-depends on A because it read one of A's updates; (2) $A \xrightarrow{ww} B$ when B write-depends on A because it overwrote one of A's updates; (3) $A \xrightarrow{rw} B$ when B anti-depends on A because A read a version of a variable for which B commits a fresher version (according to the version order $\ll$). We also refer to DSG($\mathcal{H}$) as the conflict graph.

4.4.2 Algorithm Overview

Typical MVCC algorithms [Bernstein et al., 1987] allow read-only transactions to be serialized "in the past", i.e., before the commit event of any concurrent update transaction. Conversely, they serialize an update transaction T committing at time t "in the present", by: (1) ordering versions produced by T after all versions created by transactions committed before t; (2) performing the classic validation, which ensures that the snapshot observed by T is still up-to-date considering the updates generated by all transactions that committed before t. We note that this approach is conservative, as it guarantees serializability by systematically rejecting serializable histories in which T might have been safely serialized before T'.

The key idea in TWM is to allow an update transaction to sometimes commit "in the past", by ordering the data versions it produces before those generated by already committed, concurrent transactions. In this case we say that T performs a *time-warp* commit. An example illustrating the benefits of time-warp commits is shown in Figure 4.2(a): by adopting a classic validation scheme, B would be aborted because it misses the writes issued by the two concurrent transactions A_1 and A_2; however, B can be safely serialized before both transactions that anti-depend on it, which is precisely what TWM allows for, by time-warp committing B.

To implement the time-warp abstraction efficiently, TWM orders the commit events of up-

date transactions according to two totally ordered, but possibly diverging, time lines. The first time line reflects the *natural commit order* of transactions (or, briefly, *commit order*), which is obtained by monotonically increasing a shared logical (i.e., scalar) clock and assigning the corresponding value to each committed transaction. TWM uses this time line to identify concurrent transactions and to establish the visible snapshot for a transaction upon its start. The actual transaction serialization order (and hence the version order) is instead determined by means of a second time line, which reflects what we call the *time-warp commit order* and that diverges from natural commit time order whenever a transaction performs a time-warp commit. TWM keeps track of the two time lines by associating each version of a variable with two timestamps, namely *natOrder* and *twOrder*, which reflect, respectively, the natural commit and the time-warp commit order of the transaction that created it.

We denote as $\mathcal{N}(T)$, respectively $\mathcal{TW}(T)$, the function (having the set of transactions that commit in $\mathcal{H}$ as domain, and $\mathbb{N}$ as co-domain) that defines the total order associated with the natural, respectively time-warp, commit order. Further, we write $T \prec_{\mathcal{N}} T'$, respectively $T \prec_{\mathcal{TW}} T'$, whenever $\mathcal{N}(T) < \mathcal{N}(T')$, respespectively $\mathcal{TW}(T) < \mathcal{TW}(T')$.

We start by discussing how to determine the serialization order of transactions that perform a time-warp commit. Next we describe the transaction validation logic. Finally, we explain how read and write operations are managed.

Time-warp Commit: TWM establishes the time-warp order of a committed update transaction B ($\mathcal{TW}(B)$) as follows:

***Rule* 1** *Consider that B misses the writes of a set of committed transactions A_S. This set of transactions executed concurrently with B (i.e., the transactions in A_S anti-depend on B) and committed before B (i.e., $max(\mathcal{N}(A_S)) \prec B$). Let A be the first transaction in A_S according to the natural commit order. Then $\mathcal{TW}(B) = \mathcal{N}(A)$, which effectively orders B before the transactions in A_S, namely those whose execution B did not witness. The versions of each variable updated by B are timestamped with $\mathcal{TW}(B)$ and added to the corresponding versions' chains according to the time-warp order.*

The above rule is exemplified by the history illustrated in Figure 4.2(a): as both A_1 and A_2 perform a regular commit, their time-warp order $\mathcal{TW}$ and natural commit order $\mathcal{N}$ coincide;

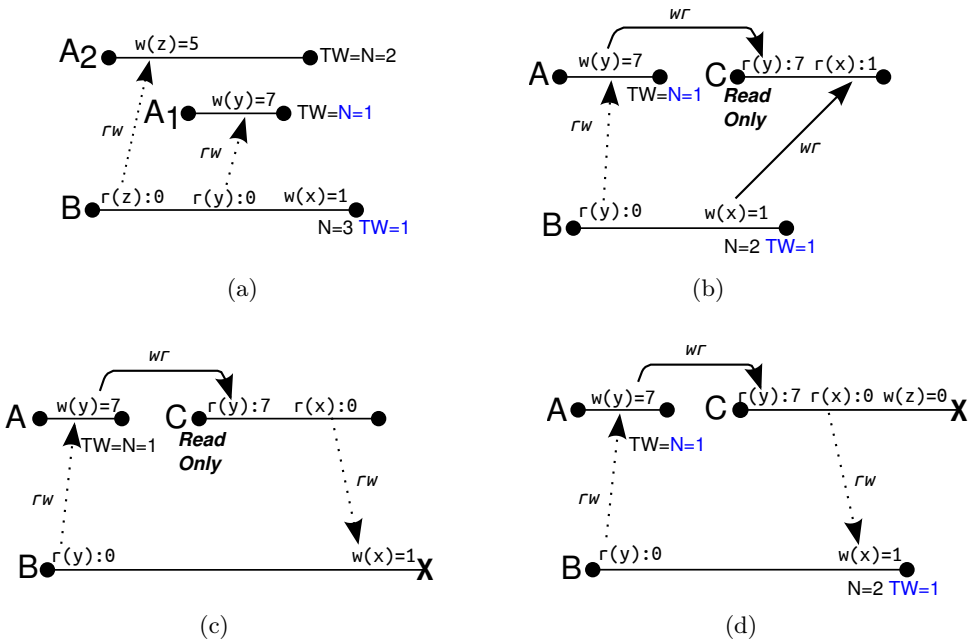


Figure 4.2: Example histories with three transactions each. The dependency edges are labelled according to the operations they connect (nomenclature of [Adya, 1999]).

conversely, as B time-warp commits due to anti-dependency edges developed towards A_1 and A_2, then B is serialized by TWM before A_1 (which commits before A_2 according to $\mathcal{N}$), and is assigned a serialization order $\mathcal{TW}(B) = \mathcal{N}(A_1) = 1$.

Validation Rule: As we will see shortly, the version visibility rule of read-only transactions ensures that these can always be correctly serialized, without the need for any validation phase. Update transactions, conversely, undergo a validation scheme that aims at detecting a specific pattern in the DSG, named *triad*. A triad exists whenever there is transaction T that is both the source and target of anti-dependency edges from two transactions T' and T'' that are concurrent with T (where, possibly, $T' = T''$, i.e., there are only two mutually anti-dependent transactions). We call T a *pivot*, and define the TWM validation scheme as follows:

***Rule* 2** *A transaction fails its validation if, by committing, it would create a triad whose* pivot *time-warp commits.*

In other words, TWM deterministically rejects schedules in which two conditions must happen: 1) a pivot transaction T misses the updates of a concurrent transaction T'; and 2) a concurrent transaction T'' (possibly T') misses in its turn the updates of the pivot transaction

T. Note that the first condition corresponds to the classic validation rule, and that the second condition (which restricts the set of histories rejected by TWM) is what allows to reduce spurious aborts with respect to state of the art STMs. Note also that a pivot must be an update transaction because a read-only transactions cannot be the target of an anti-dependency.

Figure 4.2(c) exemplifies Rule 2: when B is validated during its commit phase, TWM detects that B is the *pivot* of a triad including also A and C, and it would have to time-warp commit before A. Consequently, B is aborted; this history is indeed non-serializable. Note that B reaches that conclusion by checking solely the direct anti-dependencies it developed, which means that TWM can still generate some spurious aborts, for instance in case C did not issue $r(y)$ in Figure 4.2(c)[1]. The TWM design choice of avoiding to check indirect dependencies (i.e., dependencies developed by transactions different from the one being validated) aims to seeks a pragmatical trade-off between the cost of validating transactions and the number of serializable histories that it rejects.

Read and Write operations: It remains to discuss how TWM regulates the execution of read and write operations. Write operations are privately buffered during transaction's execution phase, and are applied only at commit time, in case the transaction is successfully validated. To determine which versions of a variable a transaction should observe, TWM attributes to a transaction, upon its start, the current value of the shared logical clock. We call this value the start of a transaction, $\mathcal{S}(T)$. TWM uses distinct version visibility rules for read-only and update transactions:

***Rule* 3** *If a read-only transaction T issues a read operation on a variable x, it returns the most recent version of x (according to the time-warp order) created by a transaction T', such that $\mathcal{TW}(T') \leq \mathcal{S}(T)$. If T is an update transaction, it is additionally required that $\mathcal{N}(T') \leq \mathcal{S}(T)$. This prevents update transactions from observing versions produced by concurrently time-warp committed transactions.*

The rationale underlying the choice of using different visibility rules for read-only and update transactions is of performance nature. TWM is designed to guarantee that read-only transactions

[1]This is unavoidable unless one performs expensive checks for cycles in the entire DSG. A detailed discussion on the set of histories accepted by TWM, based on the Input Acceptance framework [Gramoli et al., 2010] can be found in Section 4.6.

are never aborted. As a consequence, in order to preserve correctness, TWM must ensure that the snapshot observed by a read-only transaction T includes all transactions serialized before T, including time-warp committed ones (see transaction C in Figure 4.2(b)). The trade-off is that, in order to be sheltered from the risk of abort, a read-only transaction T must perform visible reads to ensure that concurrent update transactions can detect anti-dependencies originating from T (necessary to implement Rule 2). The intuition is that the combination of the triad-based validation rule and of the visible reads guarantees that an update transaction T can only time-warp commit provided that no concurrent read-only transaction has read any of the items that T wrote. Figure 4.2(c) shows a scenario in which the read-only transaction C commits and, using visible reads, allows pivot B to detect a potential violation of Rule 2, and, hence, to abort.

On the other hand, adopting visible reads for update transactions would not render them immune to aborts. Hence, TWM spares them from the cost of visible reads during their execution. Conversely, TWM adopts a lightweight approach ensuring that the snapshot visible for an update transaction T is determined upon its start, and prevent it from reading versions created by concurrent transactions that time-warped. This guarantees that the snapshot observed by T is equivalent to one producible by a serial history defined over a subset of the transactions in $\mathcal{H}$, even if T aborts. Note though that according to Rule 3 update transactions are also allowed to read the versions committed by a transaction that time-warp committed, provided that the two transactions are not concurrent.

4.4.3 Pseudo-Code Description

We now present an algorithm implementing the proposal of Time-Warp. In order to ease presentation, we begin by showing a lock-based algorithm, which we later extend to guarantee lock-freedom in Section 4.7. The pseudo-code of the lock-based algorithm is reported in Algorithms 2-5. In Table 4.2 we describe the metadata and structures used in the pseudo-code for ease of readability.

Any transaction tx starts by reading the global logical clock (*globalClock*), which defines $\mathcal{S}(tx)$. In the READ operation we first check for a read-after-write by the same transaction. Otherwise, if the reader is a read-only transaction, it registers the read in the *readers* set associated with the variable, see line 11. After that, the read-only transaction synchronizes with potential concurrent writers, by checking the lock of the variable, with the objective of making sure that

Table 4.2: Data-structures used in the TWM algorithms that follow.

Struct	Attribute	Description
`Var`	latestVersion	pointer to the most recent `Ver` of this `Var`
	readers	set of `Tx`s that read this variable
	lock	lock associated with the variable
`Ver`	value	the value of the version
	natOrder	timestamp of the *natural commit order* of the version
	twOrder	timestamp of the *time-warp order* of the version
	prevVersion	pointer to the version overwritten by this one
`Tx`	writeTx	false when this `Tx` is identified as read-only
	readSet	not used in read-only `Tx`
	writeSet	not used in read-only `Tx`
	start	timestamp of the *globalClock* when this `Tx` started
	source	true when another transaction anti-depends on `Tx`
	target	true when `Tx` anti-depends on another transaction
	natOrder	timestamp of the *natural commit order* of this `Tx`
	twOrder	timestamp of the *time-warp order* of this `Tx`

either the writer sees the visible read or that the read-only waits for the writer to conclude its commit phase. To conclude the read operation, we iterate through the versions ordered by $\mathcal{TW}$ until a condition is satisfied that reflects Rule 3.

Note that a read-only transaction that commits at time t can be removed from the *readers* sets when there is no update transaction U alive that started at time t or after. This is a process analogous to the garbage collection of multi-versions such as that employed in JVSTM [Fernandes and Cachopo, 2011] and SMV [Perelman et al., 2011], which TWM implements as well, as discussed in Section 4.4.4. This mechanism is not strictly needed for correctness and, as such, it is not detailed in the pseudo-code to avoid unnecessarily cluttering its presentation.

As mentioned before, TWM avoids any validation for read-only transactions, which always return immediately and successfully from the commit procedure. For update transactions, the COMMIT function starts by validating the writes and reads according to Rule 2 in lines 58-59.

To validate each write (function HANDLEWRITE), transaction tx checks whether there existed a concurrent transaction that read the variable since tx started, meaning there is an anti-dependence from that reader to tx. When validating a read (function HANDLEREAD), the update transaction first registers in the *readers* of the variable (similar to what read-only transactions do during execution). Then tx is said to be the source of an edge if tx read a variable and there exists a version for it that was committed after tx started. This means that such version was not

Algorithm 2: Simple lock-based STM using TWM (1/4).

```
1:  BEGIN(Tx tx, boolean isWriteTx):
2:    tx.start ← globalClock                          ▷ corresponds to S(tx)
3:    tx.natOrder ← ⊥                                 ▷ transaction is alive
4:    tx.writeTx ← isWriteTx

5:  READ(Tx tx, Var var):
6:    if tx.writeTx
7:      if ∃ ‹var, value› ∈ tx.writeSet
8:        return value                                ▷ tx had already written to var
9:      tx.readSet ← tx.readSet ∪ var                 ▷ performed by update txs
10:   else
11:     var.readers ← var.readers ∪ tx                ▷executed by read-only txs
12:     wait while is-locked(var.lock)                ▷ ensure a concurrent committer sees the visible read
13:   Ver version ← var.latestVersion
14:   while (version.twOrder > tx.start) ∨            ▷ rule 3 for read-only tx
          (tx.writeTx ∧ version.natOrder > tx.start)  do   ▷ ...and for update tx
15:     if (tx.writeTx ∧ version.natOrder ≠ version.twOrder)
16:       abort(tx)                                   ▷ early abort update tx due to rule 2
17:     version ← version.prevVersion
18:   return version.value

19: WRITE(Tx tx, Var var, Value val):
20:   tx.writeSet ← (tx.writeSet \ ‹var, _› ) ∪ ‹var, val›
```

in the snapshot of tx and thus an anti-dependency exists from tx to the transaction (say B) that produced that version. In such case, tx tries to *time-warp* commit and serialize before B. In the case that B had time-warp committed — that is known if B's time-warp and natural commit order differ from each other — then tx now fails to commit (as exemplified in Figure 4.2(d) with transaction C conducting the validation). In that case, note that B had time-warp committed, so if tx now committed as well, B would become a *pivot* breaking Rule 2. This check is also performed for update transactions during the read operation (line 16) in order to early abort them.

Also note that each anti-dependency, of which tx is the source, is stored locally during the commit procedure (line 51). This is used to implement the *time-warp* commit according to Rule 1 (see line 66). At this point tx aborts only if it raised both flags (*source* and *target* in line 61 and exemplified by Figure 4.2(c) with B conducting the validation). Otherwise, $\mathcal{N}(tx)$ is computed by atomically incrementing the global clock and reading it. The new writes are committed and stamped with both $\mathcal{TW}(tx)$ (as its version) and $\mathcal{N}(tx)$ (as the time at which it was created). Function CREATENEWVERSION places each committed write in the list of versions by using $\mathcal{TW}(tx)$ to establish the order. Because this order is non-strict, there may occur *time-*

Algorithm 3: Simple lock-based STM using TWM (2/4).

```
21: createNewVersion(Tx tx, Var var, Value val):
22:   Ver newerVersion ← ⊥
23:   Ver olderVersion ← var.latestVersion
24:   while tx.twOrder < olderVersion.twOrder
25:     newerVersion ← olderVersion
26:     olderVersion ← olderVersion.prevVersion
27:   if tx.twOrder = olderVersion.twOrder
28:     return                          ▷ no tx will ever read this value, skip it
29:   Ver version ← ‹val, tx.natOrder, tx.twOrder, tx, olderVersion›
30:   ▷ insert according to time-warp order...
31:   if newerVersion = ⊥
32:     var.latestVersion ← version                    ▷ ...as the latest version
33:   else
34:     newerVersion.prevVersion ← version             ▷ ...or as an older version
```

Algorithm 4: Simple lock-based STM using TWM (3/4).

```
35: handleWrite(Tx tx, Var var):          ▷ check if tx is the target of an edge
36:   acquire-lock(var.lock)
37:   ▷ lines 11, 12 and 36 ensure readers are either visible to tx or blocked waiting
38:   for all T_i ∈ var.readers
39:     ▷ detect concurrent transactions that read var
40:     if T_i.natOrder ≥ tx.start ∨ T_i.natOrder = ⊥
41:       tx.target ← true
```

warp clashes between transactions, i.e., $A =_{\mathcal{TW}} B$. For a set of transactions that *time-warp clash* and write to the same variable k, createNewVersion keeps only the update to k of the transaction T that has the least value for $\mathcal{N}$ (the other transactions execute line 28). In other words, the transactions in a *time-warp clash* are serialized in the inverse order of $\mathcal{N}$, because the one that happened earlier according to the natural commit order was missed by all others in the clash.

4.4.4 Garbage Collection and Privatization

Garbage Collection: The *time-warp* commit mechanism does not raise particular issues for the garbage collection of versions. Indeed, it can rely on standard garbage collection algorithms for MVCC schemes that maintain any version that can possibly be read by an active transaction (as in different implementations in [Lu and Scott, 2013, Fernandes and Cachopo, 2011, Perelman et al., 2011]). The key idea of those algorithms is the following: assume that T is the oldest active transaction, with $\mathcal{S}(T) = k$; then versions up to (and excluding) k can be garbage collected — note that the newest version is preserved regardless of this condition.

Algorithm 5: Simple lock-based STM using TWM (4/4).

```
42: HANDLEREAD(Tx tx, Var var):                          ▷ check if tx is the source of an edge
43:   ▷ tx can now do visible reads without affecting its validation
44:   var.readers ← var.readers ∪ tx
45:   wait while is-locked(var.lock) by tx′ ≠ tx
46:   ▷ check writes committed concurrently to tx's execution
47:   Ver version ← var.latestVersion
48:   while version.natOrder > tx.start
49:     if version.natOrder ≠ version.twOrder
50:       abort(tx)                                      ▷ rule 2
51:     tx.antiDeps.add(version.natOrder)                ▷ used to compute TW(tx)
52:     tx.source ← true
53:     version ← version.prevVersion

54: COMMIT(Tx tx):
55:   if !tx.writeTx
56:     return                                           ▷ read-only txs never abort
57:   ▷ check for rw edges from/to concurrent txs
58:   ∀var ∈ tx.writeSet do: HANDLEWRITE(tx, var)
59:   ∀var ∈ tx.readSet do: HANDLEREAD(tx, var)
60:   if tx.target ∧ tx.source
61:     abort(tx)                                        ▷ rule 2
62:   tx.natOrder ← incAndFetch(globalClock)             ▷ compute N(tx)
63:   if (tx.antiDeps = ∅)
64:     tx.twOrder ← tx.natOrder                         ▷ TW(tx) = N(tx)
65:   else
66:     tx.twOrder ← min(tx.antiDeps)                    ▷ compute TW(tx)
67:   ∀ ‹var, value› ∈ tx.writeSet do:
68:     CREATENEWVERSION(tx, var, value)
69:     release-lock(var.lock)
```

One may argue that a problematic scenario may arise if some update transaction U *time-warp committed* such that $\mathcal{TW}(U) < k$. For that to happen, there must exist some transaction Z concurrent with U such that: $U \xrightarrow{rw} Z$ and $\mathcal{N}(Z) < k$. But this is impossible because we assumed that T was the oldest active transaction, so Z could not be concurrent with U and obtain *natural commit order* k. This clarifies the intuition behind the fact that garbage collection algorithms remain unchanged even when using *time-warp*.

Privatization Safety: Another relevant concern is that of privatization safety [Marathe et al., 2008]. This implies that a thread should be able to safely make some shared data only available to it (i.e., to privatize it) by using a transaction P and work on it without transactional barriers (i.e., instrumentation) after committing P. The challenge here is to ensure that concurrent transactions do not interfere with the privatizing thread once is has committed P. Similarly to the concern of garbage collection, *time-warping* does not present additional challenges to

privatization.

Existing approaches to support privatization are based on the notion of *quiescence*, which forces privatizing transactions such as P to wait for concurrent transactions to finish (using, if possible, explicitly identified privatizing operations to avoid waiting when unnecessary). Such a solution is adopted, for instance, in a recent multi-versioned STM [Lu and Scott, 2013]. The idea at the basis of these techniques is to ensure that, once a privatizing transaction P has committed at time t, then all concurrent transactions with P have also committed. As all transactions starting after time t will be serialized after P (and will thus not be able to access the data privatized by P) and given that all transactions concurrent with P have concluded before P's commit is finalized, any access to the data privatized by the tread that executed P can be guaranteed to occur in absence of concurrency.

Note that these techniques are not affected by time-warping. The quiescence point defines that P holds concurrent transactions from starting until all previous ones are finished. Whether any of the running transaction time-warps only affects the serialization order. But regardless of the serialization order, after the quiescence point, all new transactions will have a starting timestamp $\geq t$. As such, the quiescence still suffices for privatization safety in TWM.

4.5 Correctness Arguments

In this section we provide arguments on the safety of the Time-Warp proposal, according to the Rules presented in Section 4.4.2, which were instantiated in the lock-based TWM algorithm in Section 4.4.3.

We begin by discussing the serializability of committed transactions in TWM, by showing that the serializability graph of histories accepted by the TWM algorithm is acyclic. Next, we discuss the consistency guarantees provided also to non-committed transactions, namely Virtual World Consistency [Imbs and Raynal, 2012].

4.5.1 Rejecting Non-Serializable Histories

In order to prove that TWM ensures serializability, we first define a strict total order ($\mathcal{O}$) on the transactions in the committed projection of $\mathcal{H}$ (noted $\mathcal{H}|C$), and then we show that any

edge between two transactions in DSG($\mathcal{H}|C$) is compliant with $\mathcal{O}$. The strict total order $\mathcal{O}$ is obtained from the non-strict total order defined by $\mathcal{TW}$, which we recall can have ties in presence of time-warp clashes, breaking ties as follows. We order update transactions in $\mathcal{O}$ using the *time-warp order* and, whenever there is a *time-warp clash*, i.e., $A =_{\mathcal{TW}} B$, we use the natural commit order $\mathcal{N}$ as a tie breaker and serialize B before A in $\mathcal{O}$ iff $A \prec_{\mathcal{N}} B$. This results in a strict total order because $\mathcal{N}$ defines a strict total order as well. Any read-only transaction T is serialized in $\mathcal{O}$ according to $\mathcal{S}(T)$, which surely makes them coincide with some update transaction in $\mathcal{O}$. To tie-break, we place the read-only transactions always later than coinciding update transactions in $\mathcal{O}$. If two read-only transactions obtain the same value (because they started on the same snapshot), any deterministic function suffices as a tie break (for instance, the identifier of the thread that executed the transaction).

Lemma 4 *TWM accepts only serializable histories.*

Proof In order to prove the acyclicity of DSG($\mathcal{H}|C$), we show that for any committed transactions A and B such that $A \prec_{\mathcal{O}} B$, there cannot be any edge from B to A in the DSG. We demonstrate this claim by contradiction, considering individually each type of edge from A to B.

First, let us assume that $B \xrightarrow{ww} A \in$ DSG($\mathcal{H}|C$). According to function createNewVersion this is possible iff $B \prec_{\mathcal{TW}} A$. This, however, directly contradicts the initial assumption $A \prec_{\mathcal{O}} B$, because it implies that $A \preccurlyeq_{\mathcal{TW}} B$.

Now let us consider that $B \xrightarrow{wr} A$. First suppose that A is an update transaction. Then, according to line 14, A can read a version created by B iff $\mathcal{N}(B) \leq \mathcal{S}(A)$. However, the time-warp commit timestamp of a transaction is always less or equal than its natural commit timestamp ($\mathcal{TW}(B) \leq \mathcal{N}(B)$); also, an update transaction A can only time-warp due to concurrent transactions, meaning they commit after $\mathcal{S}(A)$ and thus $\mathcal{S}(A) < \mathcal{TW}(A)$. Hence, we obtain $\mathcal{TW}(B) \prec \mathcal{TW}(A)$, contradicting the initial assumption. Now consider that A is a read-only transaction. Then, according to line 14, A can read a version created by B (concurrent with A's execution) iff $\mathcal{TW}(B) \leq \mathcal{S}(A)$. Given that A is a read-only transaction, $\mathcal{TW}(A) = \mathcal{S}(A)$, hence $\mathcal{TW}(B) \leq \mathcal{TW}(A)$. The case $\mathcal{TW}(B) < \mathcal{TW}(A)$ clearly contradicts the initial assumption. If $\mathcal{TW}(B) = \mathcal{TW}(A)$, then we note that A is a read-only transaction that clashes with B; according to the rules we used to derive $\mathcal{O}$ then A is ordered after B in $\mathcal{O}$, which again contradicts the

initial assumption ($A \prec_{\mathcal{O}} B$).

Finally we consider that $B \xrightarrow{rw} A$. First assume that B is a read-only transaction. Then the version written by A is not visible to B iff $\mathcal{S}(B) < \mathcal{TW}(A)$. But since B is read-only, then $\mathcal{S}(B) = \mathcal{TW}(B)$, and we once again contradict the initial assumption. Assume now that B is an update transaction, for which we have two possible cases depending on whether B commits before or after A in the *natural commit order*. Consider the first case where $B \prec_{\mathcal{N}} A$. Then B performs some visible read in line 44; later A triggers the condition in line 40 and sets $A.target \leftarrow true$. Consequently A cannot *time-warp commit* or else both *target* and *source* flags would be true and A would abort in line 61. Then $\mathcal{TW}(B) < \mathcal{N}(A) = \mathcal{TW}(A)$, which is a contradiction with the initial assumed order. Lastly, consider the second case where $A \prec_{\mathcal{N}} B$. Then B triggers the condition in line 48. If A *time-warp commits*, then B aborts in line 50. Otherwise, B adds A to it's *antiDeps* set which results in $\mathcal{TW}(B) \leq (\mathcal{N}(A) = \mathcal{TW}(A))$ (according to line 66). The case where $B \prec_{\mathcal{TW}} A$ trivially violates our initial assumption. The tie-break in the *time-warp clash*, where $B =_{\mathcal{TW}} A$, is broken in the inverse *natural commit order* (recall that $A \prec_{\mathcal{N}} B$), which also contradicts the initial assumption.

Hence, no matter which edge type we consider, it is always possible to reach a contradiction with the assumption on the order $\mathcal{O}$. Therefore TWM accepts only serializable histories.

4.5.2 Virtual World Consistency

So far we have argued that TWM ensures Serializability for committed transactions. But running (or already aborted) transactions are equally important in TWM because certain phenomena must be prevented with regard to them [Guerraoui and Kapalka, 2008, Imbs and Raynal, 2012]. The importance of this matter has also been highlighted in Section 2.2.1: if a transaction executing alone is correct, then it should be correct when faced with concurrency under a TM algorithm. This translates to a sense of consistency sufficiently strong in which hazards, such as infinite loops or divisions by zero, are avoided even for transactions that need to be eventually aborted.

The TWM algorithm guarantees such correctness, by ensuring Virtual World Consistency (VWC), which we briefly recall as being stronger than Serializability, as it prevents transactions (even those that abort) from observing snapshots that cannot be generated in any sequential

history. More precisely, besides Serializability for committed transactions[2], VWC also requires that, for every aborted or running transaction T, there is a legal linear extension of the partial order *past(T)*, where *past(T)* is obtained from the sub-graph of DSG($\mathcal{H}$) containing all the transactions on which T transitively depends, and removing any anti-dependencies. A legal linear extension of *past(T)* is a linear extension $\widehat{S}(T)$ of *past(T)* where every transaction $T' \in past(T)$ observes values written by the most recent transaction that precedes T' in $\widehat{S}(T)$.

Theorem 5 *TWM guarantees Virtual World Consistency.*

Proof By Lemma 4 we proved the absence of cycles in DSG($\mathcal{H}|C$). Now we note that *past(T)* is a subgraph of DSG($\mathcal{H}$), on which non-committed transactions are also considered; but they must be sinks in that sub-graph (because anti-dependencies are removed) and thus we also argue that *past(T)* is acyclic. It then follows that a linear extension $\widehat{S}(T)$ of *past(T)* must exist. $\widehat{S}(T)$ is legal because transactions read the most recent version committed according to $\mathcal{TW}$ (see line 14). But, since *past(T)* respects the $\mathcal{TW}$ order, we get that T must be legal and so we prove that TWM provides VWC.

Another similar, albeit stronger, correctness criterion is that of Opacity [Guerraoui and Kapalka, 2008]. In the following we discuss why TWM does not guarantee Opacity, and then explain how TWM might be adapted to ensure this property. Briefly, the Opacity specification requires 2 properties: O.1) the existence of an equivalent serial history $\mathcal{H}_\mathcal{S}$ that preserves the real-time order of $\mathcal{H}$; O.2) that every transaction in $\mathcal{H}_\mathcal{S}$ is legal. In other words, it requires that every transaction in $\mathcal{H}_\mathcal{S}$ (even if aborted) always observe, upon a read operation, the version generated by the latest write that occurred before it in $\mathcal{H}_\mathcal{S}$. In the following we show that TWM guarantees O.1, but not O.2.

Lemma 6 *TWM preserves the real-time order of $\mathcal{H}$ in the equivalent serial history $\mathcal{H}_\mathcal{S}$ that it produces.*

Proof Assume by contradiction that $\mathcal{N}(A) \prec \mathcal{S}(B)$, and that TWM serializes B before A. Then it must be that B time-warp committed and was serialized before A, namely $B \preccurlyeq_{\mathcal{TW}} A$.

[2]In fact the definition of VWC [Imbs and Raynal, 2012] allows either Serializability or strict Serializability for committed transactions. As we show in Lemma 6, TWM does indeed provide strict Serializability, as it preserves the real-time order of transactions in the equivalent sequential order that it produces besides providing Serializability (as shown in Lemma 4).

In order for B to undergo such a time-warp commit, there must exists a transaction Z such that $B \xrightarrow{rw} Z \wedge Z \prec_{\mathcal{N}} A$. In other words, B must anti-depend on Z, despite the fact that Z finishes before B starts. In such a case, however, by Rule 3, since B started in real time order after the commit event of Z, it cannot miss a value produced by Z, or, in other words, Z cannot anti-depend on B. Consequently, we reach an absurd, and prove, by contradiction, the Lemma.

On the other hand, TWM does not guarantee property O.2, as it allows two concurrent transactions R and W to perceive two different serialization orders — this is a consequence of the different version visibility conditions in line 14 according to the nature of the transaction. These two orders, denoted respectively as $\mathcal{H}_S^R$ and $\mathcal{H}_S^W$ for transactions R and W, exist in case a third concurrent transaction A time-warp commits before R and W. In this case, A may be included in $\mathcal{H}_S^R$ but not included in $\mathcal{H}_S^W$. But then, in such case, TWM would abort W due to line 50, thus not endangering Serializability. Then, this makes $\mathcal{H}_S^W$ a legal sequential history, but it is incompatible with the serial history equivalent to $\mathcal{H}$, which we denoted as $\mathcal{H}_\mathcal{S}$.

We stress that, the fact that $\mathcal{H}_S^R$ and $\mathcal{H}_S^W$ may not be compatible, is acceptable by VWC. This is because any transaction in $\mathcal{H}_S^W$ that is not compatible with $\mathcal{H}_\mathcal{S}$ aborts, and in VWC aborted transactions can observe legal linear extensions of different causal pasts. We also remark that it would be indeed relatively simple to adapt TWM to ensure property O.2, and hence Opacity: it would be sufficient to homogenize the logic governing the execution of read operations for both read-only and update transactions, allowing update transactions to observe the snapshots generated by concurrent transactions and forcing them to use visible reads, just like read-only transactions. As discussed in Section 4.4, the choice of using non-visible reads for update transactions is motivated by performance considerations. Indeed, by adopting VWC rather than Opacity as reference correctness criterion, it is possible to maximize its efficiency via lightweight conflict tracking mechanisms, while still providing robust guarantees concerning the avoidance of unexpected errors due to inconsistent/partial reads.

In fact, thanks to Lemma 6, we can actually prove a slightly stronger correctness criterion for TWM, named Strong VWC (SVWC). This criterion differs from VWC in that it requires Strict Serializability instead of Serializability for committed transactions (i.e., those transactions must respect real-time order).

Theorem 7 *TWM guarantees Strong Virtual World Consistency.*

Proof This follows straightforwardly from Theorem 5 and Lemma 6.

4.6 On the Power of TWM to Reduce Spurious Aborts

In this section we seek to assess theoretically the power of our TWM algorithm to reduce spurious aborts. For this we shall use the theory of Input Acceptance [Gramoli et al., 2010], which we first introduced in Section 2.2.2 and of which we provide a summary in the next Section 4.6.1. Afterwards, we conduct our analysis in Sections 4.6.2-4.6.6, and present the respective conclusions in Section 4.6.7.

4.6.1 Background on Input Acceptance

The key idea of input acceptance [Gramoli et al., 2010] is to establish the relative power of TMs to reduce the amount of spurious aborts. To do so, we must work with sequences of input events (composing input patterns) that, when fed to a TM, cause the abort of at least one transaction. In such case the input pattern is rejected by the TM. If the pattern results in no aborts, then it is accepted. An input class is a set of input patterns, which is rejected only when all its input patterns are rejected. Conversely, a class is accepted only when all the patterns it comprises are accepted.

We now briefly summarize the notation used in the Input Acceptance framework [Gramoli et al., 2010]. Its idea is to describe sequences of transactional data accesses using events Γ_t^x, where Γ denotes a read (r) or write (w) event of transaction t over datum x. Start and commit input events are respectively denoted by s, c. Moreover, an event π^* means any of the above (including itself, π^*) and | is the choice operator allowing a set of alternative inputs. To simplify the new classes that we present next, we add to the previous notation the following: we assume the inputs are well formed, i.e., all the data access events of a transaction T_i are preceded by a unique s_i event and followed by a unique c_i event; this means that a commit event c_i cannot occur as an expansion of a π^*, if there is a later event Γ_i^x in the input sequence. We also assume that there are no redundant events, i.e., a transaction does not perform the same access twice: if there are two reads to the same variable, only the first one is over shared memory, and the second re-uses a cached value of the first read; if there are two writes to the same variable, the first is cached locally and only the latter one is applied to the STM library call. Further, we use the

permutation operator ε as a suffix for a sequence of events to indicate any possible permutation of those events. This permutation operator cannot violate the well-formedness of expressions, i.e., it cannot permute a c_i event after a Γ_i^x event. As such, $(r_i^x\ w_j^x\ \pi^*)^\varepsilon$ means an input with at least: T_i reading x and T_j writing x, possibly with other events (in replacement of π^*), and in any order (due to the permutations allowed by ε).

By comparing the input class of different TM algorithms (also called designs, in this context), it is possible to define a hierarchy of TMs that captures their relative ability of avoiding spurious aborts. We note that, in contrast, it would not be possible to achieve such a fine-grained classification by using solely the permissiveness concept [Guerraoui et al., 2008]. To the best of our knowledge, the only opaque-permissive (online) TM is AbortsAvoider [Keidar and Perelman, 2009]. Every other online TM is simply not permissive. Yet, in practice there exist significant performance differences between such non-permissive TMs.

In the following sections we shall devise input patterns that capture the sequences of operations that lead some class of STM to abort at least one transaction. Under the Input Acceptance framework we can then create a partially ordered hierarchy of classes of STMs $\prec_{\mathcal{IA}}$ in which $STM_i \prec_{\mathcal{IA}} STM_k$ if STM_k rejects only a strict subset of the input patterns that STM_i rejects. In other words, to prove that $STM_i \prec_{\mathcal{IA}} STM_k$ one needs to show that the input class rejected by STM_k, denoted as $\mathcal{C}_k$, is strictly contained in the input class rejected by STM_i, denoted as $\mathcal{C}_i$. To help understand the intuition between each inclusion step $C_k \subset C_i$ in our proofs, we present figures of executions instantiated from the patterns; it is generally easier to understand the intuition behind the inclusion by looking at those examples. Note that, for the moment, we shall use only update transactions. We consider also transactions pre-declared as read-only, for optimization purposes of avoiding their aborts (such as in the case of TWM), in the last part in Section 4.6.7.

4.6.2 Invisible Writes Invisible Reads (IWIR)

We now start to characterize different TM designs, beginning with the Invisible Writes Invisible Reads (IWIR) design, which is characterized by read operations that do not modify shared memory and by write operations that are buffered in private memory until the transaction is deemed successful (e.g., in TL2 [Dice et al., 2006]).

$$\mathcal{C}_3 = \pi^*(r_p^x\ w_j^x\ \pi^*)^\varepsilon\ (c_j\ \pi^*)^\varepsilon\ c_p\ \pi^*$$

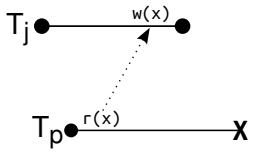


Figure 4.3: Class $\mathcal{C}_3$ and a history output by a IWIR TM for the pattern shown.

In the work of [Gramoli et al., 2010], it was shown that all the inputs rejected by IWIR are captured in the patterns of a class named $\mathcal{C}_3$. We present this class in Figure 4.3, along with an example of a possible history output by a IWIR TM when fed with an input pattern included in the class $\mathcal{C}_3$. The essence of this class is to capture the classic validation that ensures data items read are still up-to-date at commit time. As we can see, $\mathcal{C}_3$ enforces only the existence of a single anti-dependence between concurrent transactions $T_p \xrightarrow{rw} T_j$, where T_j commits before T_p (i.e., $T_j \prec_{\mathcal{N}} T_p$).

The other designs corresponding to Visible Reads Visible Writes (VWVR) (e.g., SXM [Guerraoui et al., 2005a]), and Visible Writes Visible Reads (VWIR) (e.g., TinySTM [Felber et al., 2008]), were shown to have a lower input acceptance than IWIR in [Gramoli et al., 2010], so we use the latter as a baseline in our work.

4.6.3 Time-Warp Multi-version

To study TWM our approach is to identify all conditions that can cause a transaction to abort. We then devise an input class that captures the patterns representing those conditions, i.e., the patterns rejected by TWM.

To simplify the presentation of the input acceptance of TWM, we do not consider the abort in line 16, because it is an optimization for an early detection of transactions that would, otherwise, eventually abort at commit-time. In particular, transactions that abort in line 16 would also abort in line 50 because the verified assertions are the same in both cases (i.e., version.natOrder $>$ tx.start $\wedge$ version.natOrder $\neq$ version.twOrder) and versions that existed in the former case will also exist in the latter (due to the garbage collection assumption, as stated in Section 4.4.4). As such, this does not change the input acceptance of the algorithm.

Definition The following three sets of conditions (**A**, **B**, **C**) characterize all possible aborts in

TWM (as presented in Algs. 2-5):

- **A)** In line 50 of Algorithm 5 of TWM, an update transaction T_i is aborted if it previously read some variable x and now it finds a version committed concurrently by a time-warped transaction T_p. In other words, there is a triad $T_i \xrightarrow{rw} T_p \xrightarrow{rw} T_j$. Furthermore, because T_p time-warp committed due to T_j, it follows from Rule 1 of TWM that $T_j \prec_{\mathcal{N}} T_p$. For the same reason, it follows that $T_p \prec_{\mathcal{N}} T_i$. An example of a history for this input is shown in Figure 4.4(a).

- In line 61 of Algorithm 5 an update transaction T_p aborts because it is the pivot of a triad and would have to time-warp commit (note that in **A** the aborting transaction, T_i, is not the pivot). This can happen in two cases:

  - **B)** There is a triad $T_i \xrightarrow{rw} T_p \xrightarrow{rw} T_j$. Furthermore, because T_p would time-warp commit due to T_j, then it follows from Rule 1 that $T_j \prec_{\mathcal{N}} T_p$. Also, T_p notices the visible read of T_i to y in lines 38-41, meaning that r_i^y precedes c_p. An example of a history for this input is shown in Figure 4.4(b).
  - **C)** Or there exists a degenerate triad $T_i \xrightarrow{rw} T_p \xrightarrow{rw} T_i$. For the same reason as above, it follows that $T_i \prec_{\mathcal{N}} T_p$. An example of a history for this input is shown in Figure 4.4(c).

■

To map these conditions into the notation used in the Input Acceptance framework we rely on the following mechanical transformations:

	Anti-**D**ependence Rule ($\mathbf{AD}_{j,p}$)	**N**atural **C**ommit Rule ($\mathbf{NC}_{j,p}$)
conditions	$T_p \xrightarrow{rw} T_j$	$T_j \prec_{\mathcal{N}} T_p$
input expressions	$\left(r_p^x \; w_j^x \; \pi^*\right)^{\varepsilon}$	$c_j \; \pi^* \; c_p$

These two rules for transformations follow the definitions of the anti-dependence (Section 4.4.1) and natural commit order (Section 4.4.2). In addition to that, we add a π^* event to each transformation to allow the generation of inputs with other arbitrary restrictions (i.e., events that are well-formed). The Rule AD assumes that the events c_p and c_j shall be explicit

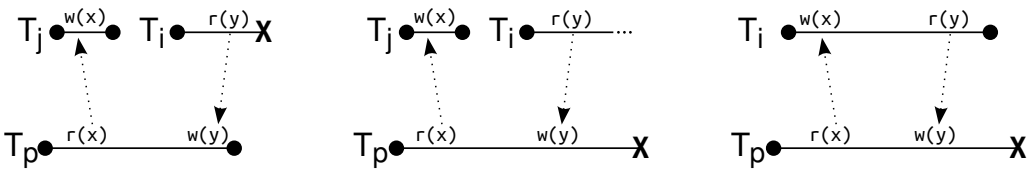


Figure 4.4: The three scenarios in which TWM aborts a transaction. For the sake of presentation, we omit variants of each scenario, in which the read/write events within a transaction may occur in different orders and moments.

$$\mathcal{C}_{tw_A} = \left(r_p^x\ w_j^x\ \pi^*\right)^{\varepsilon}\ \left(c_j\ r_i^y\ w_p^y\ \pi^*\right)^{\varepsilon}\ c_p\ \pi^*\ c_i$$
$$\mathcal{C}_{tw_B} = \left(r_p^x\ w_j^x\ \pi^*\right)^{\varepsilon}\ \left(c_j\ r_i^y\ w_p^y\ \pi^*\right)^{\varepsilon}\ c_p$$
$$\mathcal{C}_{tw_C} = (r_p^x\ w_i^x\ r_i^y\ w_p^y\ \pi^*)^{\varepsilon}\ c_i\ \pi^*\ c_p$$
$$\mathcal{C}_{tw} = \mathcal{C}_{tw_A} \mid \mathcal{C}_{tw_B} \mid \mathcal{C}_{tw_C}$$

Figure 4.5: Class $\mathcal{C}_{tw}$ is composed by three sub-classes, for each of which there is an instantiation example in Figure 4.5, respectively.

in any input pattern/class that uses this transformation, which prevents π^* from expanding as neither of them, and thus ensures that r_p^x cannot take place after c_j (because the two transactions are concurrent). We also add the ε permutation operator to allow the operations to occur in any order, which is irrelevant for the anti-dependence definition because writes are private until the commit is performed successfully.

Using the previous Definition 4.6.3 on the aborts of TWM, and the mechanical transformation of conditions into the Input Acceptance language, we now synthesize the input class $\mathcal{C}_{tw}$, shown in Figure 4.5, and prove that it captures the three sets of conditions aforementioned.

Lemma 8 *Class $\mathcal{C}_{tw}$ captures all the patterns rejected by TWM.*

Proof To prove this we use the mechanical transformations introduced before to derive, for each one of the sets **A**, **B**, and **C** presented in Definition 4.6.3, the corresponding transcription using the notation of the Input Acceptance framework.

The conditions in **A** require the triad $T_i \xrightarrow{rw} T_p \xrightarrow{rw} T_j$ and the natural order of commits $T_j \prec_{\mathcal{N}} T_p \prec_{\mathcal{N}} T_i$. Using the transformation rules we obtain the input class $\mathcal{C}_{tw_A}$:

transformation rules	$\mathrm{AD}_{j,p}$	$(\mathrm{NC}_{j,p}\ \mathrm{AD}_{p,i}\ )^{\varepsilon}$	$\mathrm{NC}_{j,p}\ \mathrm{NC}_{p,i}$
input class $\mathcal{C}_{tw_A}$	$\left(r_p^x\ w_j^x\ \pi^*\right)^{\varepsilon}$	$\left(c_j\ r_i^y\ w_p^y\ \pi^*\right)^{\varepsilon}$	$c_p\ \pi^*\ c_i$

The conditions in **B** require the triad $T_i \xrightarrow{rw} T_p \xrightarrow{rw} T_j$, the natural commit order $T_j \prec_{\mathcal{N}} T_p$ and that r_i^y precedes c_p. We capture this in the input class $\mathcal{C}_{tw_B}$:

transformation rules	$\mathrm{AD}_{j,p}$	$(\mathrm{NC}_{j,p}\ \mathrm{AD}_{p,i}\ )^{\varepsilon}$	$\mathrm{NC}_{j,p}$
input class $\mathcal{C}_{tw_A}$	$\left(r_p^x\ w_j^x\ \pi^*\right)^{\varepsilon}$	$\left(c_j\ r_i^y\ w_p^y\ \pi^*\right)^{\varepsilon}$	c_p

Finally set **C** requires the degenerated triad $T_i \xrightarrow{rw} T_p \xrightarrow{rw} T_i$ and natural commit order of $T_i \prec_{\mathcal{N}} T_p$. We capture this in the input class $\mathcal{C}_{tw_C}$:

transformation rules	$(\mathrm{AD}_{i,p}\quad \mathrm{AD}_{p,i})^{\varepsilon}$	$\mathrm{NC}_{i,p}$
input class $\mathcal{C}_{tw_C}$	$(r_p^x\ w_i^x\ r_i^y\ w_p^y\ \pi^*)^{\varepsilon}$	$c_i\ \pi^*\ c_p$

We can now use this result to start placing TWM in the hierarchy with respect to the IWIR design.

Theorem 9 *TWM has a larger input acceptance than the IWIR design.*

Proof By Gramoli et al. [Gramoli et al., 2010] we have that $\mathcal{C}_3$ captures all inputs rejected by the IWIR design. By Lemma 8 we have that $\mathcal{C}_{tw}$ captures all patterns rejected by TWM.

To prove this Theorem we are left with showing that $\mathcal{C}_{tw} \subset \mathcal{C}_3$. To do so, we show that each of the sub-classes of $\mathcal{C}_{tw}$, namely $\mathcal{C}_{tw_A}$, $\mathcal{C}_{tw_B}$ and $\mathcal{C}_{tw_C}$, can be obtained by restricting $\mathcal{C}_3$. Consequently, $\mathcal{C}_{tw}$ generates a strict subset of the input patterns of $\mathcal{C}_3$.

Let us start with $\mathcal{C}_{tw_A}$:

original $\mathcal{C}_3$	π^*	$(r_p^x\ w_j^x\ \pi^*)^{\varepsilon}$	$(c_j\ \pi^*)^{\varepsilon}$	$c_p\ \pi^*$
add restrictions to obtain $\mathcal{C}_{tw_A}$		$(r_p^x\ w_j^x\ \pi^*)^{\varepsilon}$	$\left(c_j\ \boldsymbol{r_i^y\ w_p^y\ \pi^*}\right)^{\varepsilon}$	$c_p\boldsymbol{\pi^* c_i}$

We now repeat the application of the restrictions to obtain $\mathcal{C}_{tw_B}$:

original $\mathcal{C}_3$	π^*	$(r_p^x\ w_j^x\ \pi^*)^\varepsilon$	$(c_j\ \pi^*)^\varepsilon$	$c_p\ \pi^*$
add restrictions to obtain $\mathcal{C}_{tw_B}$		$(r_p^x\ w_j^x\ \pi^*)^\varepsilon$	$\left(c_j\ \boldsymbol{r_i^y}\ \boldsymbol{w_p^y}\ \boldsymbol{\pi^*}\right)^\varepsilon$	c_p

And finally perform the same to obtain $\mathcal{C}_{tw_C}$:

original $\mathcal{C}_3$	π^*	$(r_p^x\ w_j^x\ \pi^*)^\varepsilon$	$(c_j\ \pi^*)^\varepsilon$	$c_p\ \pi^*$
renamed j to $\boldsymbol{i}$	π^*	$(r_p^x\ w_{\boldsymbol{i}}^x\ \pi^*)^\varepsilon$	$(c_{\boldsymbol{i}}\ \pi^*)^\varepsilon$	$c_p\ \pi^*$
added restrictions		$\left(r_p^x\ w_i^x\ \boldsymbol{r_i^y}\ \boldsymbol{w_p^y}\ \boldsymbol{\pi^*}\right)^\varepsilon$	$c_i\ \pi^*$	c_p
reordering of events within $^\varepsilon$ to obtain $\mathcal{C}_{tw_C}$		$\left(\boldsymbol{r_i^y}\ \boldsymbol{w_p^y}\ r_p^x\ w_i^x\ \pi^*\right)^\varepsilon$	$c_i\ \pi^*$	c_p

It follows that $\mathcal{C}_{tw} \subset \mathcal{C}_3$ and that TWM has a larger input acceptance than IWIR.

4.6.4 Interval-based

Several STMs, e.g., AVSTM [Guerraoui et al., 2008], TSTM [Aydonat and Abdelrahman, 2012] and IR_VWC_P [Crain et al., 2011], have implemented different variants of the Interval-Based (IB) design. In the following, rather than trying to capture all the details of an existing IB STM implementation, we consider a slightly more abstract, yet reasonably detailed, design that is common to such implementations. This is an approach employed also in previous works that rely on the Input Acceptance framework to simplify presentation and avoid delving into irrelevant implementation details.

The IB design maintains a lower and upper bound for each transaction T ($T.lb$ and $T.ub$), which denotes the interval of possible values for $T.ser$, i.e., the serialization point for T. These bounds are initialized with $T.lb = 0$ and $T.ub = \infty$. This interval of serialization values reflects the dependencies and anti-dependencies of T: (i) when T reads a value committed by transaction B, the algorithm enforces $T.lb \leftarrow max(T.lb, B.ser)$ to reflect that B serializes before T; (ii) when some transaction A commits, for each T whose read- and write-sets intersect on (at least) a data item, it enforces $T.ub \leftarrow min(T.ub, A.ser)$ to reflect that T serializes before A.

The choice of the serialization point, given an interval of possible values, is implementation specific. To allow a concrete comparison under the Input Acceptance framework, we consider the strategy of TSTM [Aydonat and Abdelrahman, 2012]: if $T.ub = \infty$ then $T.ser = T.lb+t$, where t is the number of concurrent threads (i.e., a parameter of the system); otherwise, $T.ser = T.ub-1$.

The rationale of this strategy is to reserve enough space for concurrent transactions to serialize before T, if needed.

We now define under which conditions an IB TM aborts a transaction:

Definition A transaction T_i is aborted in the IB design if T_i's interval for serialization is empty during the commit-time validation. Otherwise T_i can choose any value between $[T_i.lb, T_i.ub]$ using the rule described above.

More formally, the conditions that lead the IB design to abort T_i are the following:

1. $\exists\ T_a : T_a \xrightarrow{wr} T_i \vee T_a \xrightarrow{ww} T_i$
2. $\exists\ T_j : T_i \xrightarrow{rw} T_j$
3. $T_j.ser < T_a.ser$

In other words: (1) requires T_i to depend on T_a, which implies both $T_a \prec_{\mathcal{N}} T_i$ and $T_a.ser < T_i.ser$; (2) requires T_i to anti-depend on T_j, which implies both $T_j \prec_{\mathcal{N}} T_i$ and $T_i.ser < T_j.ser$. Hence, together with (3), this would create a contradiction, namely $T_a.ser < T_i.ser < T_j.ser < T_a.ser$, which is why T_i is aborted (as the other two have already been assigned serialization points).

■

Next we prove an upper bound on the patterns rejected by the IB design, which will later be used to compare the input acceptance of the IWIR design. To help with this we create a new transformation rule in addition to the two others previously presented (Rules AD and NC):

	Dependency Rule ($\mathbf{DEP}_{a,i}$)
conditions	$T_a \xrightarrow{ww} T_i \vee T_a \xrightarrow{wr} T_i$
input expressions	$(w_a^y\ c_a\ \pi^*)^\varepsilon\ ((r_i^y \mid w_i^y)\ c_i\ \pi^*)^\varepsilon$

Lemma 10 *Class $\mathcal{C}_{ib}$ is an upper bound on the patterns rejected by the IB design.*

$$\mathcal{C}_{ib} = (w_a^y\ c_a\ \pi^*)^\varepsilon\ \left((r_i^y \mid w_i^y) r_i^x\ w_j^x\ \pi^*\right)^\varepsilon (c_j\ \pi^*)^\varepsilon\ c_i$$

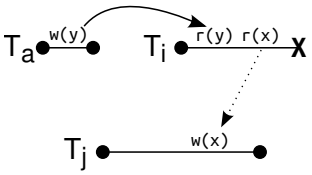


Figure 4.6: Class $\mathcal{C}_{ib}$ and an example history output by a TM using the IB design for a pattern included in the class.

Proof To prove this Lemma we show how the set of necessary conditions for aborting a transaction in IB (as presented in Definition 4.6.4) can be expressed using the notation of the Input Acceptance framework.

We ignore condition (3), which is hard to capture precisely using the Input Acceptance framework. Because of this, the result input class encompasses more patterns than the ones actually aborted by IB, and is hence an upper bound.

Conditions (1) and (2) are summarized as follows: $(T_a \xrightarrow{wr} T_i \vee T_a \xrightarrow{ww} T_i)$, and $T_i \xrightarrow{rw} T_j$. By using the two previous tables of transformation rules from conditions to input expressions, we get the upper bound on the patterns rejected by IB:

transformation rules	$\text{DEP}_{a,i}$	$\text{AD}_{j,i}$	$\text{NC}_{j,i}$
input class $\mathcal{C}_{ib}$	$(w_a^y\ c_a\ \pi^*)^\varepsilon\left((r_i^y \mid w_i^y)\right.$	$r_i^x\ w_j^x\ \pi^*\left.\right)^\varepsilon$	$(c_j\ \pi^*)^\varepsilon\ c_i$

Next, we prove that IB has a larger input acceptance than the IWIR design by showing that $\mathcal{C}_{ib} \subset \mathcal{C}_3$, where we recall that $\mathcal{C}_3$ captures the patterns rejected by the IWIR design.

Theorem 11 *The IB design has a larger input acceptance than the IWIR design.*

Proof By Gramoli et al. [Gramoli et al., 2010] we have that $\mathcal{C}_3$ captures all inputs rejected by the IWIR design. By Lemma 10 we have that $\mathcal{C}_{ib}$ is an upper bound on the patterns rejected by the IB design, hence capturing all (and possibly more than) the patterns rejected by IB.

To prove this Theorem we are thus left with showing that $\mathcal{C}_{ib} \subset \mathcal{C}_3$. To do so, we proceed similarly to the proof of Theorem 9 and add restrictions to the input class $\mathcal{C}_3$ until it becomes equal to $\mathcal{C}_{ib}$:

original $\mathcal{C}_3$	π^*	$(r_p^x\ w_j^x\ \pi^*)^\varepsilon$	$(c_j\ \pi^*)^\varepsilon$	$c_p\ \pi^*$
rename p to $\boldsymbol{i}$	π^*	$(r_{\boldsymbol{i}}^x\ w_j^x\ \pi^*)^\varepsilon$	$(c_j\ \pi^*)^\varepsilon$	$c_{\boldsymbol{i}}\ \pi^*$
add restrictions	$\boldsymbol{(w_a^y\ c_a\ \pi^*)^\varepsilon}$	$\left(r_i^x\ w_j^x\ \boldsymbol{(r_i^y\mid w_i^y)\ \pi^*}\right)^\varepsilon$	$(c_j\ \pi^*)^\varepsilon$	c_i
reordering within $^\varepsilon$ to obtain $\mathcal{C}_{ib}$	$(w_a^y\ c_a\ \pi^*)^\varepsilon$	$\left(\boldsymbol{(r_i^y\mid w_i^y)}\ r_i^x\ w_j^x\ \pi^*\right)^\varepsilon$	$(c_j\ \pi^*)^\varepsilon$	c_i

Therefore we show that $\mathcal{C}_{ib} \subset \mathcal{C}_3$ and thus conclude that IB has a larger input acceptance than IWIR.

4.6.5 Comparing TWM and IB

Theorems 9 and 11 state that both the TWM and IB designs have a larger input acceptance than IWIR. The question now is to assess the relative power of TWM and IB. In doing so, we arrive at the conclusion that they are incomparable.

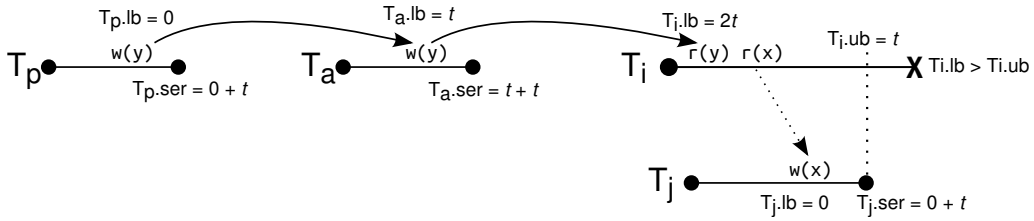
Lemma 12 *The IB design does not have larger input acceptance than TWM.*

Proof To prove this we use an input pattern, which we call $\mathcal{P}_{ib}$, and show that it is rejected by the IB design while being accepted by TWM:

input pattern $\mathcal{P}_{ib}$	$s_p\ w_p^y\ c_p\ s_a\ w_a^y\ c_a\ s_i\ s_j\ r_i^y\ r_i^x\ w_j^x\ c_j\ c_i$

One can see that this patterns is reject by IB as it can be generated from $\mathcal{C}_{ib}$, while additionally ensuring condition (3) from Definition 4.6.4. The figure below illustrates, in detail, the execution of the IB design when fed with $\mathcal{P}_{ib}$, also showing how the rule described in Definition 4.6.4 (taken from TSTM [Aydonat and Abdelrahman, 2012]) is used to assign the transaction serialization points. Without loss of generality, we assume that a special transaction T_0 initially writes every variable with a default value and serializes on point 0 before every other transaction. Note that t is the number of concurrent transactions in the system (hence ≥ 0), i.e., a parameter of the strategy for assigning serialization points. By the figure, we can see that condition (3) (i.e., $T_j.ser < T_a.ser$) holds and, hence, T_i is aborted.

When this pattern is fed to TWM, however, no transaction is aborted: by Rule 1 of TWM it is necessary for a triad to exist for a transaction to abort, and we note that $\mathcal{P}_{ib}$ contains only



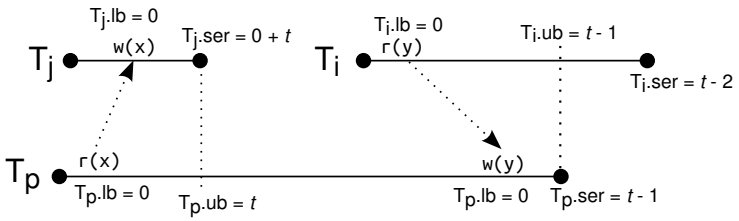
one anti-dependency, that of $T_i \xrightarrow{rw} T_j$, which is insufficient to generate a triad. Consequently TWM accepts $\mathcal{P}_{ib}$, and the Lemma is proved.

Lemma 13 *TWM does not have a larger input acceptance than the IB design.*

Proof We prove this analogously to the previous case: i.e., we find a pattern that is rejected by TWM but that is accepted by IB. For that we use an input pattern $\mathcal{P}_{tw_A}$, generated from $\mathcal{C}_{tw_A}$, which was shown to be rejected by the TWM design in Lemma 8.

input pattern $\mathcal{P}_{tw_A}$	$s_p\ s_j\ r_p^x\ w_j^x\ c_j\ s_i\ r_i^y\ w_p^y\ c_p\ c_i$

We now argue that $\mathcal{P}_{tw_A}$ is accepted by IB as it does not abort any transaction given that input pattern (contrarily to TWM, which aborts T_p as it is the pivot of a triad and it would have to time-warp commit). As in the previous Lemma, we assume that a special transaction T_0 initially writes every variable with a default value and serializes on point 0 before every other transaction. Then, we show that, when fed with this input pattern $\mathcal{P}_{tw_A}$, the IB design is able to find a serialization point for each transaction and commit all of them (with serialization order $T_i \prec T_p \prec T_j$).



As a result, IB accepts $\mathcal{P}_{tw_A}$, which is rejected by TWM and thus we prove the Lemma.

Theorem 14 *TWM and the IB design are incomparable with regard to input acceptance.*

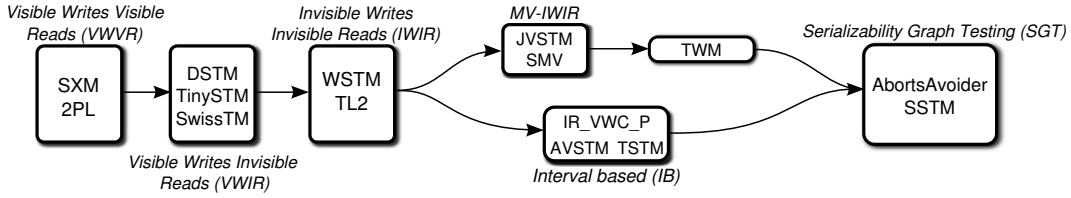


Figure 4.7: Comparison of the input acceptance of Time-Warp with other TM designs.

Proof This follows trivially from Lemmas 12 and 13, as neither approach has larger input acceptance than the other.

4.6.6 Serializability Graph Testing

Finally, we describe a design with higher input acceptance than TWM and IB. To do so, we identify a design called Serializability Graph Testing (SGT). This design draws its key idea from the Serializability Theorem [Bernstein et al., 1987]. This theorem states that a history $\mathcal{H}$ is serializable if and only if the corresponding Direct Serialization Graph DSG($\mathcal{H}$) is acyclic. The AbortsAvoider TM [Keidar and Perelman, 2009] explores this technique to ensure online-opaque-permissiveness by maintaining an explicit DSG. On the other hand, SSTM [Gramoli et al., 2010] also uses SGT, albeit it scatters the corresponding metadata across transactions. This means there is no centralized (or even explicit) DSG; but this is an implementation detail that has no impact from the perspective of input acceptance.

4.6.7 Revised Input Acceptance Hierarchy

We can finally revise the hierarchy originally presented in [Gramoli et al., 2010] according to our results, as shown in Figure 4.7. By Theorems 9 and 11 we place TWM and IB above IWIR, and by Theorem 14 we place them side by side.

So far, in our analysis, we considered solely update transactions. This was done because we compare both single-versioned and multi-versioned TM designs, where the latter can deterministically avoid aborting read-only transactions [Perelman et al., 2010]. By accounting also for read-only transactions in our input acceptance analysis, we introduce an additional design called MV-IWIR. This design includes mv-permissive TMs based on IWIR algorithms and, aside from this, uses the same validation rule to commit update transactions. Both JVSTM [Fernandes and

Cachopo, 2011] and SMV [Perelman et al., 2011] exploit this design by using invisible reads, deferred writes, and multi-versions to guarantee that read-only transactions never abort.

The only difference from IWIR to MV-IWIR is that read-only transactions do not abort. This means that, by considering only update transactions, MV-IWIR also rejects all the patterns included in $\mathcal{C}_3$. If read-only transactions are identified a priori, i.e., the start event of a read-only transaction is annotated, then MV-IWIR achieves a larger input acceptance than IWIR. The reason why TWM has a larger input acceptance than MW-IWIR is that, despite both being mv-permissive, the class of update transactions rejected by TWM is a subset of that rejected by (MV-)IWIR.

We additionally observe that IB is incomparable with MV-IWIR. On one hand, IB accepts the example execution in Figure 4.3, which is rejected by MV-IWIR if T_i is a write transaction. On the other hand, MV-IWIR accepts the example execution in Figure 4.6 if T_i is a read-only transaction, which was shown to be rejected by IB. TWM remains incomparable with IB even with read-only transactions, as the considerations concerning the existence of input patterns accepted by IB and rejected by TWM (and vice versa) still apply.

This concludes our theoretical analysis of TWM, with the resulting hierarchy justifying (together with our experimental study, which we shall present in Section 4.8) the ability of time-warping of reducing spurious aborts in TM applications.

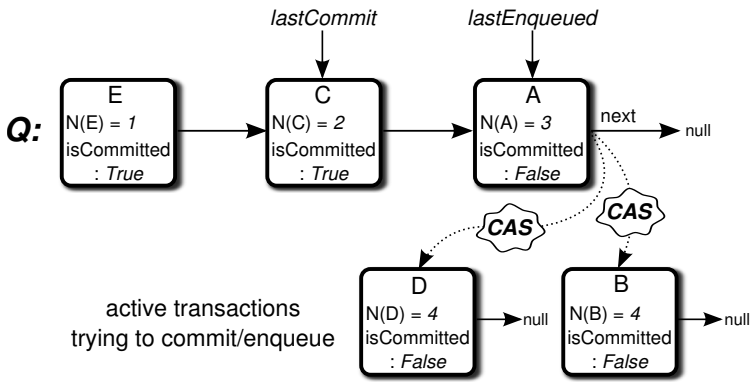


Figure 4.8: Transactions D and B trying to commit with $\mathcal{N} = 4$.

4.7 A Lock-Free Optimized Algorithm

Recent work has motivated the adoption of lock-free synchronization schemes to enhance scalability [Guerraoui et al., 2008, Fernandes and Cachopo, 2011] of TM implementations, for which reason we now devise a practical and efficient algorithm that instantiate the rules described in Section 4.4.2. The two main differences consist of the adoption of lock-free mechanisms to ensure progress in our STM, as well as an efficient technique to replace the *readers* set of every variable.

To make TWM lock-free we leveraged on the techniques presented in [Fernandes and Cachopo, 2011] and adapted them to fit TWM. The main idea is that the commit of a transaction T now considers three phases: (1) validate T; (2) enqueue T in $\mathcal{Q}$, a queue of committed (and eventually finished) transactions; and (3) ensure that every transaction in $\mathcal{Q}$ (up to and including T) finishes by applying their write-sets to shared memory. As we shall see, the queue $\mathcal{Q}$ is at the core of the lock-free commit procedure and, naturally, is implemented as a lock-free queue.

In fact, we do not use a shared global clock but, instead, use the queue to derive the *natural commit order*: when a transaction T enqueues successfully (after some transaction P) then T computes $\mathcal{N}(T)$ simply as $\mathcal{N}(P) + 1$. In other words, the *natural commit order* stems from the order acquired by each transaction in $\mathcal{Q}$. Figure 4.8 shows an example for this procedure, where transactions D and B have conducted a validation successfully and attempt concurrently to enqueue in $\mathcal{Q}$ — which explains why both tentatively compute 4 as their *natural commit order*. Naturally, only one of the enqueues shall succeed, as the other compare-and-swap will fail and the corresponding transaction shall restart the procedure.

To complement the pseudo-code description we rely on Table 4.3, which contains the structures used in our pseudo-code. In the following sections we shall walk through the Algorithms 6-10 that compose the lock-free TWM. To ease the understanding of these more complex algorithms, we highlight the line numbers that correspond to additional logic with regard to the earlier lock-based implementation.

4.7.1 Begin, Read and Write in a Transaction

Every transaction tx starts by computing its $\mathcal{S}(tx)$ during the BEGIN function, i.e., obtains a logical time that maps to a consistent snapshot of the shared memory that it can read. This

Table 4.3: Updated data structures used in the lock-free TWM, with regard to Table 4.2 — the new attributes, not used in the simple lock-based algorithm, are highlighted.

Struct	Attribute	Description
`Var`	**readStamp**	timestamp when this `Var` was last read
	latestVersion	pointer to the most recent `Ver` of this `Var`
	committingTx	pointer to a `Tx` attempting to commit to this `Var`
`Ver`	value	the value of the version
	natOrder	timestamp of the *natural commit order* of the version
	twOrder	timestamp of the *time-warp order* of the version
	prevVersion	pointer to the version overwritten by this one
`Tx`	writeTx	false when this `Tx` is identified as read-only
	readSet	not used in read-only `Tx`
	writeSet	not used in read-only `Tx`
	start	timestamp of the when this `Tx` started
	source	true when another transaction anti-depends on `Tx`
	target	true when `Tx` anti-depends on another transacton
	natOrder	timestamp of the *natural commit order* of this `Tx`
	twOrder	timestamp of the *time-warp order* of this `Tx`
	isCommitted	true when the transaction is written-back
	next	pointer to the next (more recent) `Tx` enqueued in $\mathcal{Q}$ after this `Tx`
$\mathcal{Q}$	**lastCommit**	pointer to `Tx` that was written-back (may not be the last, due to a race)
	lastEnqueued	pointer to `Tx` that was last enqueued for commit (head of the queue)

is simply derived from the *natural commit order* of the latest committed transaction in $\mathcal{Q}$. The logic embedded in lines 23-28 reflects the fact that the *lastCommit* pointer of $\mathcal{Q}$ may not be up-to-date — it may not be pointing actually to the latest committed transaction. This is due to performance reasons that shall be explained later in this document.

In the READ function we replaced the usage of the *readers* set by a semi-visible readers scheme. The semi-visible read procedure relies on a scalar associated with each variable (the attribute *readStamp*) to capture the latest global clock at which some transaction read the variable. Conceptually adding a transaction to the set of readers is implemented by applying the current clock only if it is larger than the latest visible read (line 33). This corresponds to a semi-visible read scheme because we do not track individually each reader (contrarily to more onerous approaches [Gramoli et al., 2010, Aydonat and Abdelrahman, 2012, Guerraoui et al., 2008, Keidar and Perelman, 2009]).

We also removed the lock associated with each variable. Instead, each variable has a *committingTx* attribute that may point to an update transaction attempting to commit a new value to that variable. This part of the algorithm guarantees synchronization between the read-only

Algorithm 6: Begin transaction, write and read functions for lock-free TWM.

```
1:  BEGIN(Tx tx, boolean isWriteTx):
2|    tx.start ← GETLATESTCOMMIT()                          ▷ corresponds to S(tx)
3:    tx.writeTx ← isWriteTx

4:  WRITE(Tx tx, Var var, Value val):
5:    tx.writeSet ← (tx.writeSet \ ‹var, _› ) ∪ ‹var, val›

6:  READ(Tx tx, Var var):
7:    if tx.writeTx
8:      if ∃ ‹var, value› ∈ tx.writeSet
9:        return value                                     ▷ tx had already written to var
10:     tx.readSet ← tx.readSet ∪ var                      ▷ performed by update txs
11:   else
12|     SEMIVISIBLEREAD(tx, var)                           ▷ performed by read-only txs
13|     Tx writer ← var.committingTx
14|     if writer ≠ ⊥ ∧ not writer.isCommitted
15|       COMMIT(writer)                                   ▷ help commit the writer
16:   Ver version ← var.latestVersion
17:   while (version.twOrder > tx.start) ∨                 ▷ rule 2 for read-only tx
            (tx.writeTx ∧ version.natOrder > tx.start)  do  ▷ ...and for update tx
18:     if (tx.writeTx ∧ version.natOrder ≠ version.twOrder)
19:       abort(tx)                                        ▷ early abort update tx due to rule 2
20:     version ← version.prevVersion
21:   return version.value

22| GETLATESTCOMMIT():
23|   Tx lastCommitted ← Q.lastCommit
24|   while lastCommitted.next ≠ ⊥ do
25|     Tx moreRecent ← lastCommitted.next                 ▷ obtain the actual last committed tx
26|     if moreRecent.isCommitted
27|       lastCommitted ← moreRecent
28|   return lastCommitted.natOrder

29| SEMIVISIBLEREAD(Tx tx, Var var):
30|   repeat
31|     long ts ← GETLATESTCOMMIT()
32|     long lastRead ← var.readStamp
33|   until lastRead ≥ ts ∨ CAS(var.readStamp, lastRead, ts) = success
```

transaction tx and any such possible concurrent writer to the variable being read. As such, tx invokes function SEMIVISIBLEREAD, after which it checks for a possible writer transaction that may have concurrently missed the semi-visible read.

If tx finds some *writer* transaction trying to commit an update to *var*, it helps the commit of *writer* before proceeding with the read. We must now consider what a read-only R has to do when faced with a variable undergoing commit by T (line 14). To preserve this procedure with a non-blocking nature, we make R help decide the fate of T — this is implemented as invoking COMMIT for T, which, naturally, is tailored to have concurrent threads attempting to commit the same transaction. This is the fundamental idea of transactions helping each other, which is typical in lock-free algorithms. This allows R to ensure progress for T and decide if the write of T is visible to the snapshot of R or not, and in the positive case, safely read it.

The rest of the READ function remains the same. We highlight that the condition for write transactions is more demanding, as it may iterate over more versions than read-only transactions: this is a consequence of the fact that read-only transactions perform the semi-visible read scheme, which enables them to read fresher values with the guarantee of a consistent snapshot. This avoids the need for a commit time validation, and is crucial in preserving the abort-freedom of read-only transactions.

4.7.2 Lock-Free Commit Procedure

We now enhance the commit procedure of TWM to be lock-free. For this, we highlight that the COMMIT procedure may be invoked by different threads attempting to commit the same transaction. For instance: read-only transactions that need to decide the fate of a writer, to decide on which version to read; also write transactions that try to commit, need to help transactions that obtained an earlier *natural commit* order.

This pattern of allowing transactions to help those that would impede progress is common in the literature to guarantee lock-freedom [Herlihy et al., 2003a, Herlihy, 1991, Fernandes and Cachopo, 2011]. As a result, we end up having several compare-and-swap instructions in which the outcome is not relevant: the idea is that *some* helper transaction will execute it successfully, for which reason these compare-and-swaps are not used in the typical 'loop until succeed' fashion. It should be noted that, in order to simplify presentation, in the pseudo-code we are assuming

Algorithm 7: Commit procedure for lock-free TWM.

```
34: COMMIT(Tx tx):
35:   if !tx.writeTx
36:     return                                          ▷ read-only txs never abort
37|   Tx lastCommit ← HELPCOMMITALL()
38|   tx.natOrder ← lastCommit.natOrder                ▷ not the final value for 𝒩(tx)
39:   ∀var ∈ tx.writeSet do: HANDLEWRITE(tx, var)
40:   ∀var ∈ tx.readSet do: HANDLEREAD(tx, var)
41|   COMPUTECOMMIT(tx)
42|   if tx.isCommitted ∨ 𝒬 contains tx
43|     return                                          ▷ some helper succeeded first
44|   while not CAS(𝒬.lastEnqueued, lastCommit, tx)
45|     lastCommit ← INCREMENTALVALIDATION(tx, lastCommit)
46|     tx.natOrder ← lastCommit.natOrder
47|     COMPUTECOMMIT(tx)
48|     if tx.isCommitted ∨ 𝒬 contains tx
49|       return                                        ▷ some helper succeeded first
50|   HELPCOMMITALL()                                  ▷ ensure tx is written-back

51| COMPUTECOMMIT(Tx tx):
52:   if tx.target ∧ tx.source
53:     abort(tx)                                      ▷ rule 3
54|   tx.natOrder ← tx.natOrder + 1                    ▷ compute 𝒩(tx)
55:   if (tx.antiDeps = ∅)
56:     tx.twOrder ← tx.natOrder                       ▷ 𝒯𝒲(tx) = 𝒩(tx)
57:   else
58:     tx.twOrder ← min(tx.antiDeps)                  ▷ compute 𝒯𝒲(tx)
```

that a helping thread alters directly the attributes of the `Tx` structure, whereas in an actual implementation several of the writes performed by a helping thread are issued to thread-local memory. This is simply so that concurrent helpers to the same transaction do not have to synchronize on accesses to memory that is logically private to the transaction.

At the start of the COMMIT the thread starts by helping finalize the commit of every transaction enqueued in $\mathcal{Q}$ but not yet committed. As a result, it returns the most recently known committed transaction, which will be used as the expected value for the head of $\mathcal{Q}$ in the compare-and-swap operation in line 44. Line 38 sets $\mathcal{N}(tx)$ ephemerally to the value of $\mathcal{N}$ of the last known committed transaction — this is merely a temporary value, and will be re-computed in line 54 before the transaction commits.

This commit procedure implements two different parts: lines 39-43 validate transaction tx against the most recently known committed transaction; after that the transaction is enqueued in $\mathcal{Q}$; however, if that fails, due to some concurrent transactions that got enqueued in the meantime, then an extra validation (lines 45-49) is necessary to ensure that the newly enqueued transactions

do not invalidate tx.

During the normal validation HANDLEWRITE contains an additional step that helps ensure lock-freedom. In lines 60-63 the committing thread tries to acquire ownership of the *var.committingTx* attribute for the variables written during the speculative execution (to preserve lock-freedom, this step is performed over an ordered write-set). Recall that the attribute *committingTx* is used by read-only transactions, so that they can synchronize with a potential committing writer transaction. If a helping thread fails to set the ownership to transaction T in some variable k, it must be because of a concurrent helper committing some T' with intersecting writes. In such case, T helps T' validate and enqueue in $\mathcal{Q}$ before proceeding with its own commit (line 63). As we will discuss shortly, these pointers are cleared when the variables are written-back, to allow more transactions to commit new versions of those variables.

Inserting a transaction in the queue is simply a compare-and-swap operation, in line 44, to the *next* attribute of the last transaction seen as committed (resulting from the helping mechanism). This step may fail when some (possibly set of) concurrent transaction(s) enqueued instead in that position (one at a time). In such event, T has to guarantee it is still valid, by taking into account the transactions that won the race to $\mathcal{Q}$. This is achieved by using the read- and write-sets of the transactions that enqueued concurrently (function INCREMENTALVALIDATION, see Algorithm 7). We perform a selective validation between T and each $C_i \in \{C_1, ..., C_N\}$ among the N transactions that won the race for the enqueue:

- $C_i \xrightarrow{rw} T$ exists when $T.writeSet \cap C_i.readSet \neq \emptyset$.
- $T \xrightarrow{rw} C_i$ exists when $T.readSet \cap C_i.writeSet \neq \emptyset$

Note that the first check may be skipped if T was already known to be the target of a similar edge. Besides the implementation nature, we can see that the logic governing these validations is the same as presented for the normal validation. For the second verification, we note that typically the write-set is much smaller than the read-set, which makes these verifications efficient if the contains operation of the set data structure used to implement read-set and write-set has a complexity bounded by $O(1)$.

At this point, in line 50, a lock-free helping mechanism is used to commit each transaction in $\mathcal{Q}$ that is pending finalization (which is the case for T).

Algorithm 8: Auxiliary functions for the commit in the lock-free TWM (1/3).

```
59: HANDLEWRITE(Tx tx, Var var):                    ▷ check if tx is the target of an edge
60|   while not CAS(var.committingTx, ⊥, tx)
61|     Tx otherWriter ← var.committingTx
62|     if otherWriter ≠ ⊥
63|       COMMIT(otherWriter)
64:   if var.readStamp ≥ tx.start
65:     ▷ detect concurrent transactions that read var
66:     tx.target ← true

67: HANDLEREAD(Tx tx, Var var):                     ▷ check if tx is the source of an edge
68|   SEMIVISIBLEREAD(tx, var)
69:   ▷ check writes committed concurrently to tx's execution
70:   Ver version ← var.latestVersion
71:   while version.natOrder > tx.start
72:     if version.natOrder ≠ version.twOrder
73:       abort(tx)                                                    ▷ rule 3
74:     tx.antiDeps.add(version.natOrder)                  ▷ used to compute TW(tx)
75:     tx.source ← true
76:     version ← version.prevVersion
```

4.7.3 Finishing a Transaction

The HELPCOMMITALL function simply traverses the transactions in $\mathcal{Q}$ not yet committed and attempts to help each one. The implementation of HELPCOMMIT resembles that of [Fernandes and Cachopo, 2011]. Briefly, this entails applying the contents of the write-set of tx to the respective variables, and this is performed in parallel by any transaction that concurrently tries to help by splitting the write-set in buckets — note that the pseudo-code omit these details for ease of presentation. Moreover, each helper verifies that all the buckets are processed before considering T as finished. This idea of splitting the write-set in buckets is in fact shown here by iterating over the whole write-set in line 100.

The procedure of committing a buffered value to a variable is reified in CREATENEWVERSION, and contains a subtle change only. We note that there may exist concurrent threads inserting committed values into the same version list. Hence, they may not be committing the same transaction, as it can happen that a thread t_1 helps commit T, which is finished concurrently; then some other thread committing another transaction will be concurrent with t_1 committing T. Therefore the placement of the version in the list of versions uses a compare-and-swap operation (lines 116 and 119).

Finally, the set of new versions of a commit are atomically available to new transactions

Algorithm 9: Auxiliary functions for the commit in the lock-free TWM (2/3).

```
77| incrementalValidation(Tx tx, Tx lastCommit):
78|   Tx toCommit ← lastCommit.next
79|   ▷ consider each new transaction
80|   while toCommit ≠ ⊥
81|     if tx.writeSet ∩ lastCommit.readSet ≠ ∅
82|       tx.target ← true
83|     if tx.readSet ∩ lastCommit.writeSet ≠ ∅
84|       if toCommit.source
85|         abort(tx)
86|       tx.source ← true
87|     lastCommit ← toCommit
88|     toCommit ← toCommit.next
89|   return lastCommit

90| helpCommitAll():
91|   Tx lastCommit ← Q.lastCommit
92|   Tx toCommit ← lastCommit.next
93|   while toCommit ≠ ⊥
94|     helpCommit(toCommit)
95|     lastCommit ← toCommit
96|     toCommit ← toCommit.next
97|   return lastCommit

98| helpCommit(Tx toCommit):
99|   if not toCommit.isCommitted
100|     for all ‹var, value› ∈ toCommit.writeSet
101|       createNewVersion(toCommit, var, val)
102|       CAS(var.committingTx, toCommit, ⊥)
103|     toCommit.isCommitted ← true
104|     Q.lastCommit ← toCommit
```

after line 103 is processed by at least one helper. Note that $\mathcal{Q}.lastCommit$ acts merely as a shortcut to speed the start of transactions.

4.7.4 Discussion on Lock-Freedom

In this section, we provide insights on the ability of the previous algorithms to ensure progress guarantees. In the following analysis we show that this algorithm makes TWM lock-free because it ensures that a thread only fails to achieve forward progress if some other thread does so. Also, progress guarantees are ensured independently of whether the execution of any thread in the system is suspended for arbitrary long periods of time.

The begin operation may repeat the operation to obtain the latest committed snapshot only in the event that concurrent transactions are committing. The write operation executes indepen-

Algorithm 10: Auxiliary functions for the commit in the lock-free TWM (3/3).

```
104: createNewVersion(Tx tx, Var var, Value val):
105:   Ver newerVersion ← ⊥
106:   Ver olderVersion ← var.latestVersion
107:   while tx.twOrder < olderVersion.twOrder
108:     newerVersion ← olderVersion
109:     olderVersion ← olderVersion.prevVersion
110:   if tx.twOrder = olderVersion.twOrder
111:     return
112:   Ver version ← ‹val, tx.natOrder, tx.twOrder, tx, olderVersion›
113:   ▷ insert according to time-warp order...
114:   if newerVersion = ⊥
115:     ▷ ...as the latest version
116|      CAS(var.latestVersion, olderVersion, version)
117:   else
118:     ▷ ...or as an older version
119|      CAS(newerVersion.prevVersion, olderVersion, version)
```

dently of concurrent events. For the read operation, we must consider several cases. The semi-visible read may repeat only if a concurrent transaction updated the timestamp concurrently, and if it updated it to a more recent timestamp. This means that some concurrent transaction must have committed, or otherwise no concurrent transaction could update the timestamp to a more recent one. Then, the reader may help a commit operation, which we argue next is also a lock-free procedure. Finally, the reader also eventually finds a version to read, or aborts. Hence, all these procedures eventually conclude, or only repeat when faced with a concurrent transaction that committed, meaning the system progressed.

Concerning the commit operation, this may repeat when another write transaction succeeds on enqueuing concurrently in $\mathcal{Q}$; but that means there was global progress. T may also repeat the commit procedure if it fails to place itself as the writer of some variable k, because it is already being written by some T'. But in such case, T retreats by removing itself from the writer of variables, and helps T' with validation and enqueue. Because the write-sets are canonically ordered, it is impossible for a cycle of helping dependencies to exist. Therefore there will always exist some write transaction T enqueuing successfully and ensuring global progress.

With regard to read-only transactions, they never block, and do not have to execute any commit (and of course write) procedure. The read operation of a read-only transaction R helps at most the validation and enqueue of a write transaction, and, as explained in the last paragraph, a helping operation can only fail and repeat in the presence of global progress.

4.8 Evaluation

In this section we experimentally evaluate the performance of a Java-based implementation of the lock-free algorithm of TWM presented in Section 4.7. To access its merit, we compare it with five other STMs representative of different designs, most of which we discussed in Section 2.3:

- JVSTM [Fernandes and Cachopo, 2011] is multi-versioned and guarantees abort-freedom for read-only transactions;
- TL2 [Dice et al., 2006] is a simpler TM based on timestamps and locks;
- NOrec [Dalessandro et al., 2010] uses a single word for metadata (a global lock), thus being even simpler than TL2;
- TSTM [Aydonat and Abdelrahman, 2012] is lock-based and relies on the *interval-based* approach (see Section 4.6);
- AVSTM [Guerraoui et al., 2008] is also single-version, but on top of that it is also probabilistically permissive with regard to Opacity.

This allows to contrast TWM directly against two different designs that minimize spurious aborts (AVSTM and TSTM); against TM algorithms designed to optimize efficiency at low thread counts (TL2 and NOrec); and against a multi-versioned TM (JVSTM). Note that both JVSTM and AVSTM are lock-free (similarly to our prototype of TWM, as explained in Section 4.7), whereas TL2 and NOrec are lock-based. Finally, TWM, AVSTM and TSTM exploit alternative mechanisms to validate transactions, whereas the others rely on the *classic validation* rule.

We used Java implementations for all the STMs, by obtaining the code for JVSTM from its public repository, TL2 and NOrec from their respective ports to the Deuce framework [Felber et al., 2010b], and by porting AVSTM and TSTM to Java. All implementations share a common interface that uses selective instrumentation of shared variables in the benchmarks (akin to the STAMP benchmarks) relying on a concept similar to that of `VBoxes` [Fernandes and Cachopo, 2011]. This means that the benchmarks were manually instrumented to identify shared variables and transactions, resulting in an equal and fair environment for comparison of all TMs. We also identified read-only transactions in the benchmarks, and allowed implementations to take advantage of this when possible. This means that TWM, JVSTM, TL2 and TSTM do not

Table 4.4: Characteristics of the Opteron machine, with a large number of cores, which was used to evaluate TWM.

Resource	Description
Processors	4 × AMD Opteron 6272 2.1GHz
Cores	64 (16 on each processor)
L1 Cache	16KB 4-way (per core)
L2 Cache	2MB 16-way (per 2 cores)
L3 Cache	8MB 64-way (per 8 cores)
Cache Line	64B
RAM Size	32GB
Operating System	Ubuntu 12.04
NUMA distances	60% more latency to any remote NUMA bank than to a local one

maintain read-sets for such transactions and their commit procedure needs no validation. NOrec requires the read-set for re-validation of a transaction T when the global clock has changed, and AVSTM requires it for an update transaction T that is committing to update the validity interval of concurrent transactions T' that read items committed by T.

In the following experimental study we seek to answer the following questions:

1. What is the performance difference of TWM to each of the other design class of STM?
2. Where does the difference in performance come from?
3. What is the overhead in reducing aborts with respect to the classic validation?

To answer the above questions, we conducted experiments on a variety of benchmarks and workloads. We first present results with a classic micro-benchmark for TM, namely Skip List, which allows us to study and understand in detail the differences among the STMs considered. Next we test other more complex and realistic benchmarks, namely STMBench7 [Guerraoui et al., 2007] and the STAMP suite of benchmarks [Minh et al., 2008]. STAMP typically contains smaller and less conflicting transactions than STMBench7, although every transaction contains writes (i.e., it does not contain read-only transactions), which is a disadvantage particularly for the multi-versioned TM algorithms.

The following results were obtained on a machine with four AMD Opteron 6272 processors (64 total cores), 32GB of RAM, running Ubuntu 12.04 and Oracle's JVM 1.7.0_15, which we also present in Table 4.4. Each data point corresponds to the average of 10 executions. We

use the geometric mean when we show averages over normalized result and use as abort rate metric the ratio of number of restarts to the number of executions (encompassing committed and restarted transactions).

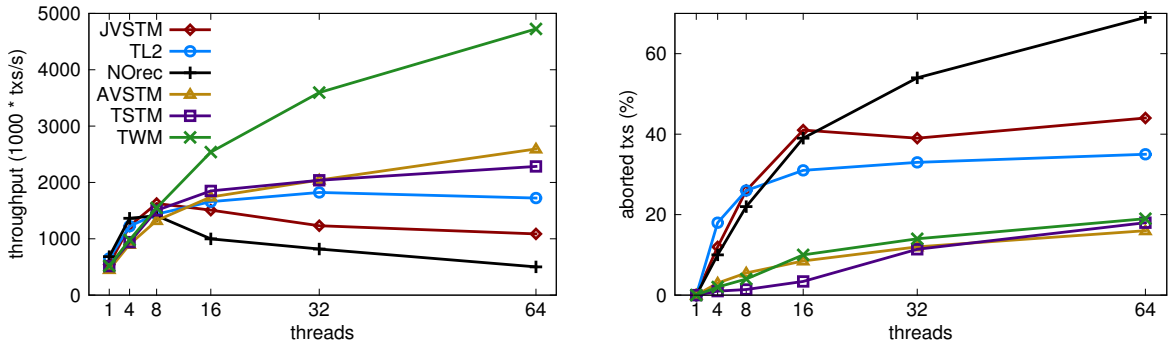
4.8.1 Skip List

We begin by studying the behavior of time-warping in a traditional data-structure. As described in Section 4.1, concurrent traversals and modifications in data-structures, such as a skip-list, are perfect examples of the advantages of time-warping: a transaction T_1 modifying an element near the end of the list need not abort only because a concurrent transaction T_2 modified an element in the beginning of the list and committed; TWM can automatically, and safely, commit T_1 before T_2, whereas *classic validation* would have precluded the commit of T_1.

For this micro-benchmark we used the source code available in the IntSet benchmark in the Deuce framework. We set up a skip-list with 100 thousand elements and 25% update transactions that either insert or remove an element. Figure 4.9(a) shows the scalability results for this workload, where TWM performs best after 8 threads, and below that is competitive with the other TMs. At 64 cores TWM achieves the following speedups: 2.8× over TL2; 9.4× over NOrec; 4.3× over JVSTM; 1.8× over AVSTM; and 2.1× over TSTM. It is actually interesting to assess that, at a low thread count, NOrec performs best. However, this difference quickly fades as the number of threads increases and its performance plunges due to the overly pessimistic validation procedure — this is visible on Figure 4.9(b) where its abort rate quickly grows to approximately 70%. Note that JVSTM's update transactions incur significant costs due to the multi-version maintenance — this cost is amplified by the non-negligible percentage of update transactions, which take no advantage from the availability of multiple versions. TWM, instead, takes advantage of multi-versioning even for update transactions thanks to time-warping.

Overall, as we can see in Figure 4.9(b), the source of our gains is two-fold: TWM clearly aborts much less transactions than TM algorithms relying on *classic validation*; on the other hand, despite TWM aborting slightly more than AVSTM, it introduces a much lower overhead, as we shall discuss in detail in the next section.

But before that, in Figure 4.10, we show the mean speedup (averaged over degree of parallelism) of TWM against the other STMs when varying the percentage of update transactions.



(a) Throughput in Skip List.

(b) Abort percentage in Skip List.

Figure 4.9: Skip List with 25% modifications.

This summarized perspective over this set of experiments allows us to see different trends of improvement. Against JVSTM, our approach continuously improves as the workload becomes more write-intensive. This is a consequence of the fact that, despite both algorithms guarantee that read-only transactions never abort, TWM further avoids spurious aborts for update transactions. For TL2 and NOrec, the trend is similar as for JVSTM, except that in write-heavy workloads TWM loses some advantage. This can be explained considering that multi-versioning no longer pays off, but, conversely, becomes a burden, in scenarios encompassing a very reduced percentage of read-only transactions. Finally, both AVSTM and TSTM become increasingly similar to TWM when increasing the ratio of update transactions, because they all reduce spurious aborts similarly and the advantage of abort-free read-only transactions is gradually lost by TWM. Among these two, TSTM seems to have a more efficient algorithm. Notably, TWM does not add any benefit against the TM algorithms using classic validation in absence of up-

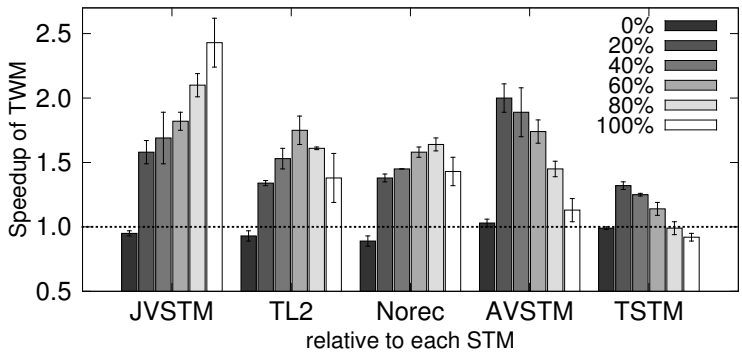


Figure 4.10: Geometric mean speedup (for all threads) of TWM relatively to each STM when varying the percentage of update transactions in Skip List.

date transactions, because of the added complexity that it introduces in order to reduce spurious aborts, which is clearly useless in abort-free scenarios.

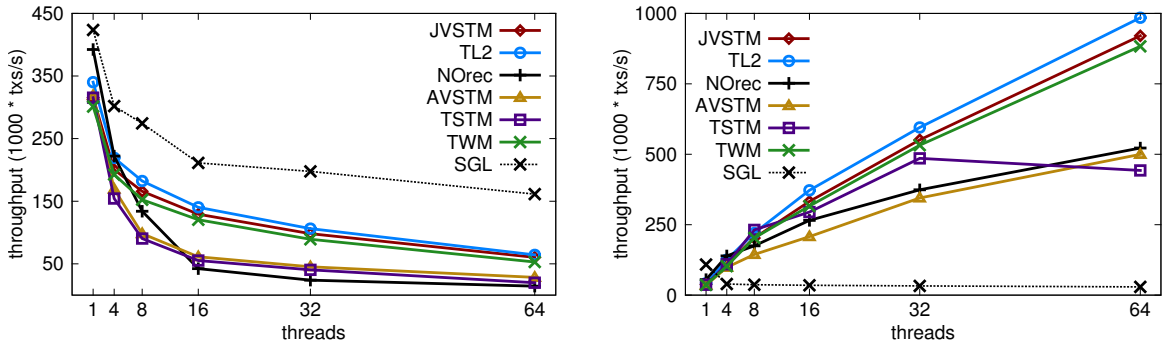
4.8.1.1 Overhead Assessment in the Skip List

In order to better understand the concomitant tradeoffs of each design, we designed worst-case experiments aimed to assess the cost of reducing spurious aborts. We also provide in these experiments variant with a single-global lock (SGL) to protect all transactions; this illustrates the overhead of instrumentation that STMs impose, which is mostly visible with low threads. We first conducted an experiment with two shared variables, both incremented once by every transaction, to create a scenario with very high contention and whose conflict patterns cannot be accommodated by the TWM algorithm (as well as by the other considered TMs).

We can see the throughput for this experiment in Figure 4.11(a), where the slowdown of TWM is comparable to that of JVSTM and TL2, being 7% and 12% worse with respect to those two TMs. The others perform worse beyond 8 threads due to the internal validation procedures — we shall discuss this phenomenon in detail next.

We also modified the Skip List micro-benchmark to have each thread modify an independent skip-list. Consequently, no transaction ever runs into conflicts, although they still activate the validation procedures as every transaction performs some writes. The results of this experiment are shown in Figure 4.11(b). As expected, every TM is able to scale as this scenario is conflict-free. The notable exception is TSTM, whose global readers table induces spurious failed compare-and-swaps in the read operations at high degrees of parallelism. Overall, the relative trends are consistent with those observed for the highly-contended scenario with the shared counters.

To gain deeper insights on these results, we instrumented the prototypes to collect the time spent by transactions in each phase of the TM algorithm. Figure 4.12 shows the results relative to the previous experiment. We considered four different phases: the *read* corresponds to time spent in read barriers; *readSet-val* and *writeSet-val* are the validations conducted by the transaction, including those at a commit-time and when executed during the transaction execution in the case of NOrec — note that the write-set validation only exists in the case of TWM, AVSTM and TSTM; and finally *commit* corresponds to the rest of the time spent in the commit phase (for instance, writing-back, or helping other transactions in the case of lock-free schemes).



(a) Incrementing two shared counters.

(b) Skip List without contention.

Figure 4.11: Overhead assessment with 100% writes.

In this plot, we see that the commit is generally the main source of overhead as the thread count increases. TL2 obtains the least overhead because transactions are conflict-free and the workload is write-intensive, which implies extra costs for schemes that minimize aborts and for multi-version algorithms. Initially, NOrec also benefits from these circumstances to yield the least overhead. However, it quickly becomes less efficient as the global commit lock becomes a bottleneck and the commit time increases significantly due to threads waiting to commit. Moreover, its read-set validation time also increases because transactions re-validate the read-set when they notice that the global clock has changed (due to the concurrent commit of an update transaction).

On the other hand, the lock-free schemes also incur in some overhead right from the start. Both TWM and AVSTM conduct additional validations that are useless in this scenario, as it is conflict-free, and are noticeably making them more expensive. Yet, TWM preserves the

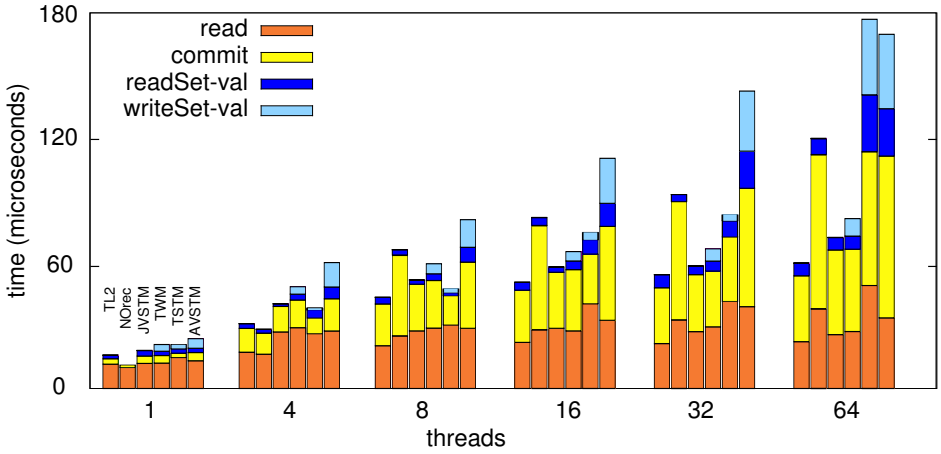


Figure 4.12: Overhead breakdown for Skip List without contention and 100% writes.

overheads rather low as the scale increases, whereas AVSTM suffers considerably as we reach 64 threads, making it the most expensive TM at that scale, slightly above TSTM. The main culprit for this cost in AVSTM is the fact that a committing update transaction must possibly update metadata of every concurrent transaction. As a result of this onerous check, the commit and validation costs grow considerably with the number of threads. The same happens for TSTM, as both algorithms explore similar interval-based approaches, with the difference that TSTM is lock-based.

TWM, similarly to JVSTM, is a multi-versioned STM. In fact, they both use the same garbage collection infrastructure, as described in Section 4.4.4, in which a version is kept in memory as long as some transaction may be able to read it. To provide a better insight into the memory cost of supporting multi-versions, we plot in Figure 4.13 the average number of versions available to a read operation over a shared variable in the Skip List benchmark. We show these numbers for varying write ratios and two levels of concurrency. In every scenario we had the garbage collection mechanism configured to run every 500ms, which is a conservative figure, so that it does not interfere with the execution of the application code.

This profiling over the multi-versions shows that the number of versions grows, as expected, with the percentage of write transactions. Furthermore, the more concurrent threads we have running transactions, the more versions we end up with until the garbage collection threads execute. This is an inherent cost for MVCC algorithms. However, as also corroborated by our experimental results, such cost is often outweighed by the benefits that multi-versioning brings about in terms of abort reduction both for read-only transactions (for conventional MVCC algorithms [Fernandes and Cachopo, 2011, Perelman et al., 2011]) and update transactions (for

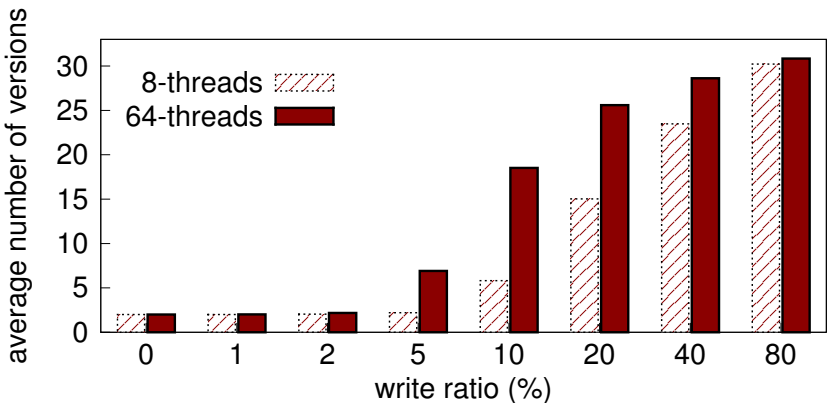


Figure 4.13: Average number of versions available to each read operation for TWM.

TWM).

Finally, we highlight that TWM's overheads are consistently close to those of JVSTM. They are also both higher than those of TL2 due to the management of multi-versions and lock-freedom guarantees. Yet, TWM manages to reduce spurious aborts with respect to both JVSTM and TL2. This illustrates the key appealing feature of TWM, namely its ability to reduce spurious aborts while incurring low instrumentation overheads even in low contention scenarios.

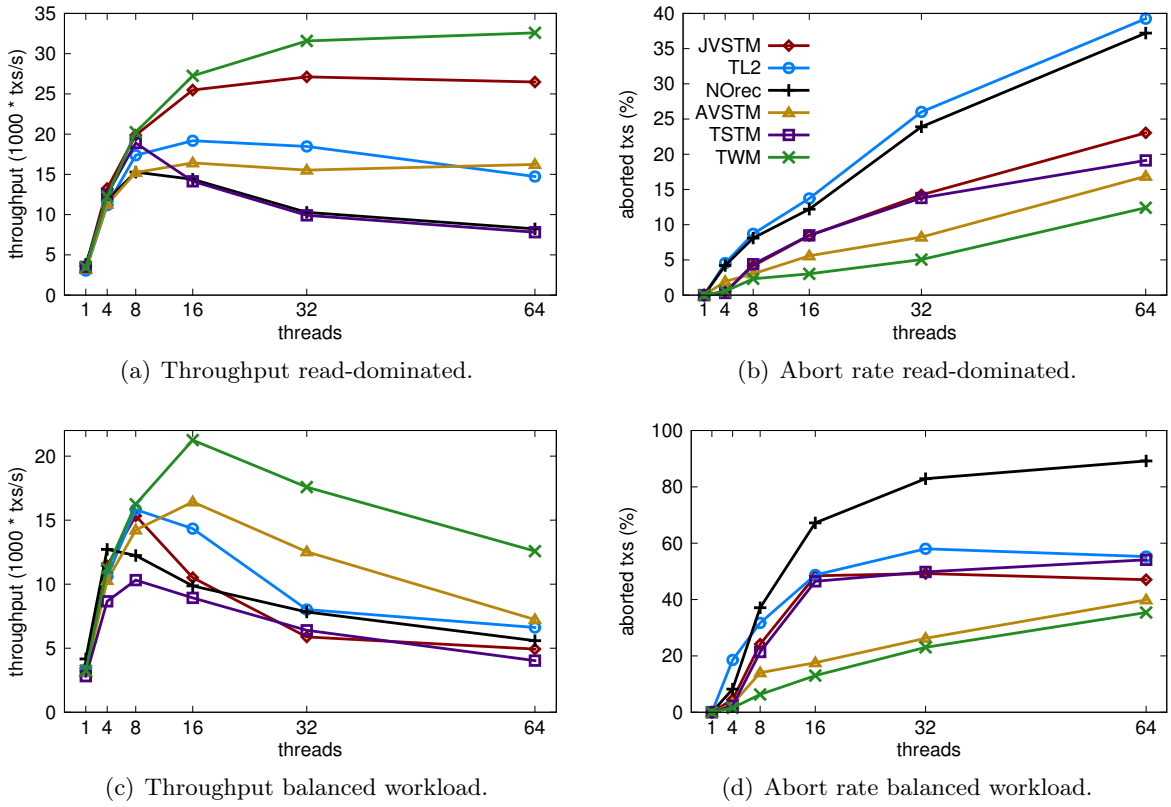
4.8.2 Macro-Benchmarks

In this section we present additional experiments using a set of larger and more complex benchmarks. This allows us to determine whether the TWM performance gains that we highlighted in the previous section using micro-benchmarks, are still confirmed when considering more complex (and arguably more realistic) applications.

STMBench7. We begin with STMBench7 with structural modifications enabled and two workloads, with 90 and 50% read-only transactions. This benchmark, and in particular the latter configuration, leads to a significant amount of contention.

We start by looking at the read-dominated workload shown in Figure 4.14(a). At a minimum of 8 threads, we can already see the difference that it makes to reduce spurious aborts. Abort reduction is imputable to two reasons: read-only transactions that do not abort, as is the case for JVSTM and TWM; as well as update transactions that abort less, which happens for TWM, AVSTM and TSTM. Naturally, the joint reduction in both dimensions allows TWM to obtain the best performance. Figure 4.14(b) shows the abort rate for this scenario, where we can see that TWM has the lowest abort rate across all degrees of parallelism. Given the dominance of read-only transactions, this also justifies the good performance of JVSTM, contrasting with that of the other TMs that reduce spurious aborts for update transactions at the cost of aborting read-only transactions (AVSTM and TSTM).

For the balanced workload, in Figure 4.14(c), the landscape changes considerably. Despite the presence of read-only transactions, the high contention makes it very difficult for any TM to scale. This is clear from the very high abort rates that the considered TM algorithms generally experience. Namely, for the TM algorithms that use *classic validation* we can see that performance starts dropping at 8 threads. TWM and AVSTM, instead, scale further to 16 threads.



(a) Throughput read-dominated.

(b) Abort rate read-dominated.

(c) Throughput balanced workload.

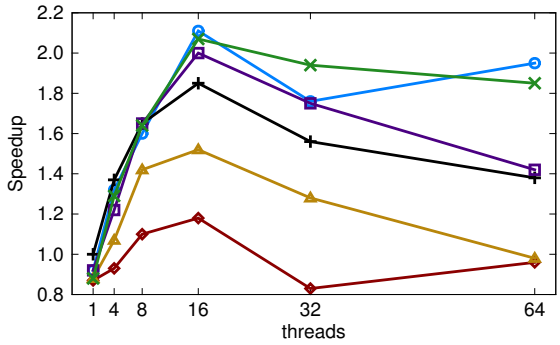
(d) Abort rate balanced workload.

Figure 4.14: STMBench7 workloads.

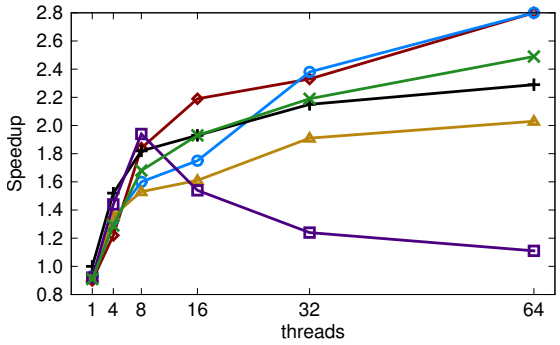
TWM achieves an average speedup of 1.37×, 1.46×, 1.57×, 1.26×, and 1.9× with respect to TL2, NOrec, JVSTM, AVSTM, and TSTM. As shown in Figure 4.14(b), the gains obtained by TWM versus the other considered STM algorithms are due to reduced spurious aborts. The only exception being AVSTM, which achieves abort rates similar to TWM, but which suffers from a more costly algorithm as shown in the previous section.

STAMP. We have also studied the performance of these TM algorithms in STAMP, for which we used an existing port to Java that is available in the Deuce framework. Figure 4.15 presents the speedup relatively to a sequential execution of NOrec, which is consistently the fastest one with a single thread. We have excluded the benchmarks Yada (not available in the Java porting) and Bayes (excluded given its non-determinism).

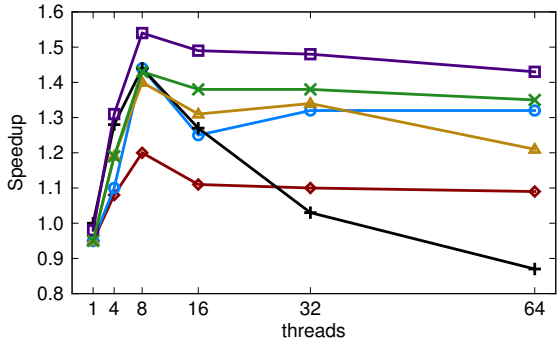
TWM behaves slightly worse than JVSTM and TL2 in both Intruder and Kmeans with an average slowdown of 7%. On the other benchmarks, it is either on par with the best TM (Genome, SSCA2, Vacation-low) or it obtains improvements over all TMs (Labyrinth and Vacation-high). We have manually inspected each benchmark to understand if there are opportunities for time-



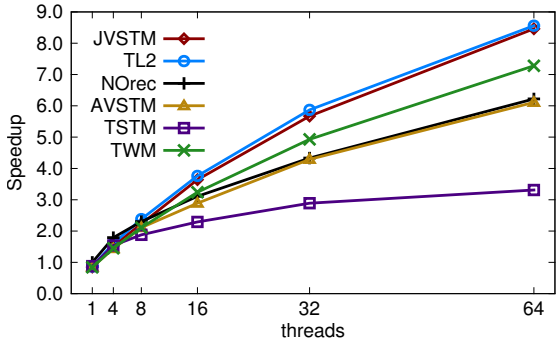
(a) Genome.



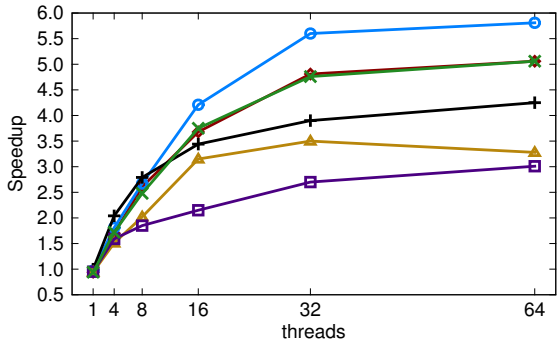
(b) Intruder.



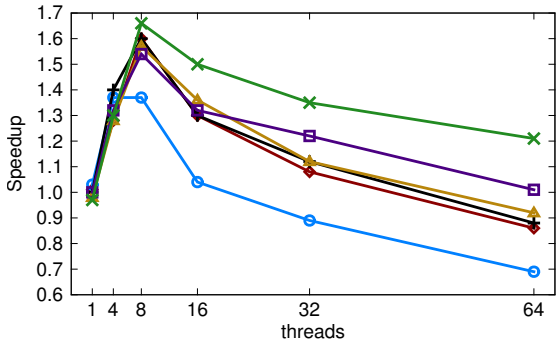
(c) SSCA2.



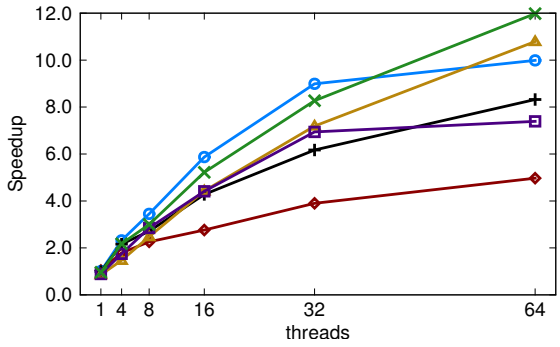
(d) Kmeans (low).



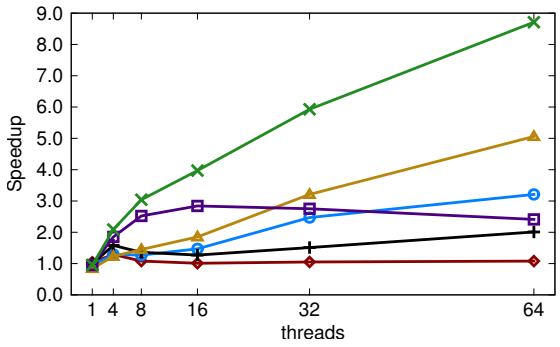
(e) Kmeans (high).



(f) Labyrinth.



(g) Vacation (low).



(h) Vacation (high).

Figure 4.15: Scalability in the STAMP benchmarks.

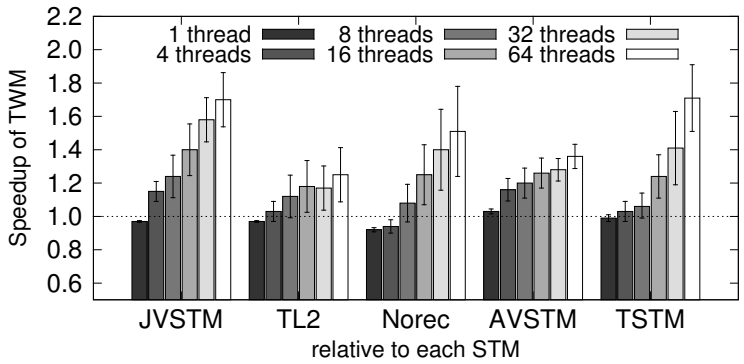


Figure 4.16: Average speedup of TWM relatively to each STM for all STAMP benchmarks across different numbers of threads.

warp to reduce spurious aborts: this is the case for Genome, Labyrinth and Vacation. The other three only generate simple conflict patterns that cannot be avoided with time-warping. Yet, TWM manages to perform among the best TM algorithms in every benchmark.

Conversely, AVSTM only obtains considerable improvements in Vacation (high), although it still performs worse than TWM. TSTM usually obtains a good initial performance, but its blocking nature and the overhead of a global readers table prevent its scalability. The latter is in contrast with the semi-visible readers scheme used by TWM.

Next, we present a plot summarizing these experiments in Figure 4.16. There, we show the geometric mean of the speedup of TWM relative to the other TM algorithms across all the STAMP benchmarks. Notice that we are varying the degree of parallelism in each bar of the plot. The overall trend is that TWM is more beneficial than TM algorithms that use *classic validation* as the thread count increases. The average improvement across all the benchmarks is 31% over JVSTM, 12% over TL2, 16% over NOrec, 21% over AVSTM and 25% over TSTM. Additionally, if we only consider the benchmarks with possibility of time-warping, TWM obtains an average improvement of 36% over JVSTM, 37% over TL2, 41% over NOrec, 37% over AVSTM, and 35% over TSTM.

Note that the gains over AVSTM and TSTM are mostly due to the fact that time-warp based validation mechanism allows for reducing spurious aborts in a more efficient way than interval-based schemes, as for most benchmarks these TM algorithms achieve a very similar abort rate (as shown in Table 4.5). In order to back this claim, we additionally present Table 4.5, where we show the average abort rate per thread count (averaging all benchmarks) and per benchmark

Table 4.5: Average abort rate (%) across each STAMP benchmark (above) and each thread count (below).

	Benchmark							
STM	genome	intruder	kmeans-l	kmeans-h	labyrinth	ssca2	vac-l	vac-h
TWM	3.8	3.8	1.4	4.2	8.8	10.5	6.4	17.8
JVSTM	15.4	3.2	1.6	4.9	12.3	11.3	12.1	41.1
TL2	12.1	4.8	3.8	3.4	13.8	11.7	10.0	41.4
NOrec	21.1	6.0	3.8	6.4	27.6	14.9	19.9	55.0
AVSTM	13.0	3.5	2.6	4.8	10.4	11.5	9.4	18.9
TSTM	12.1	3.2	0.9	4.1	9.9	9.1	5.1	18.2

	Threads				
STM	4	8	16	32	64
TWM	1.2	4.4	6.6	9.9	15.7
JVSTM	1.8	7.0	10.2	15.7	21.2
TL2	2.6	6.5	11.4	16.1	20.9
NOrec	3.4	9.6	18.6	24.9	34.0
AVSTM	2.5	5.5	8.6	12.7	17.6
TSTM	1.0	5.1	7.4	11.4	20.3

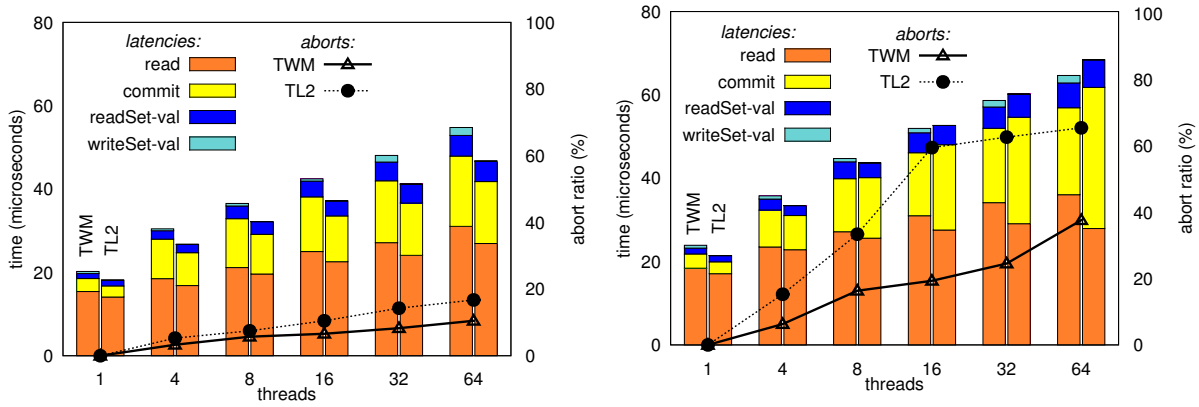
(averaging all thread counts). Generally, we can see that the abort rate of TWM, AVSTM and TSTM are similar. Finally, these data confirm that not all STAMP benchmarks can actually benefit from the time-warping mechanism to reduce spurious aborts.

4.8.2.1 Overhead assessment in the Application Benchmarks

The previous evaluation has shown that TWM is quite competitive for most scenarios but it shines particularly when there is high contention and a high number of threads. One of the applications where this is most visible is Vacation, from the STAMP benchmark, whose contrasting scenarios illustrate this in Figs. 4.15(g) and 4.15(h): in low contention, TWM outperforms TL2 only when using 64 threads, whereas in high contention it yields the best performance starting from 4 threads.

In order to shed lights on the reasons underlying these performance figures, we report in Figure 4.17 the execution times of successfully committed transactions with TWM and TL2 in both Vacation scenarios. We additionally show the abort ratios of each STM in each configuration under evaluation.

In general, across almost all the scenarios, we can see that the difference in the transaction execution time of TWM and TL2 is roughly constant (similarly to what we had shown before



(a) Low contention.

(b) High contention.

Figure 4.17: Overhead breakdown with abort ratios for TWM and TL2 in Vacation.

in Figure 4.12) and around 10%. The only exception to this rule is the case of high contention in Vacation, in which the transaction execution in TL2 is worse than in TWM at high levels of concurrency. Although the difference is small, it is visible in Figure 4.17(b), where essentially TL2's commit increases slightly due to the memory contention to acquire the same locks. TWM is not subject to this issue do its lock-free implementation in which a thread committing cannot hold the others back.

More relevant are instead the differences between the two TM implementation in terms of abort ratio. In fact, we can see that TWM takes the lead in performance as the abort rate in TL2 reaches at least 20% -this happens at 64 threads for low contention and when using at least 4 threads for high contention. In these scenarios, we can see that the abort rate with TWM is at least 10% lower than for TL2. Hence, in these scenarios the benefits achieved by TWM in terms of abort reduction outweigh the additional overheads that it introduces - which, as discussed above, can be also approximately quantified around 10%.

4.9 Distributed Time-Warping

As described earlier, in Sections 2.3 and 2.6, the ideas behind STMs translate quite naturally to a distributed setting (where they are called DTMs). In this section we succinctly describe how we applied the core ideas of Time-Warp to a genuine partially replicated DTM (i.e., a type of DTMs that scale to large settings — recall Section 2.6) and complement this with a summary of the evaluation study that we conducted on this new setting.

4.9.1 Overview of the Distributed Time-Warping Protocol

The motivation in this setting is very similar to that of STMs, as DTMs operate in a similar way to ensure correctness of concurrent transactions, in practice using also the classic validation rule. This is indeed the case for SCORe [Peluso et al., 2012b], a state of the art 1-Copy Serializable [Adya, 1999] DTM that we extended with Distributed Time-Warping (DTW). We recall that SCORe was initially described in Section 2.6.

In SCORe each transaction executes only in nodes (i.e., independent processes possibly in distributed machines) that replicate data accessed, so that it preserves the genuineness property, and thus allows to scale performance as more machines are added — assuming that each transaction accesses only a small portion of data replicated in a sub-set of the nodes. As such this is an important property that we seek to preserve. This is a perfect matching to the core idea of time-warping, as it keeps track only of the direct conflicts of each transaction: as such, we must only add the metadata necessary for time-warping, which in this case will execute distributed, on each node where the transaction ran, to identify conflicts on the sub-set of the data accessed there.

In this way, the key mechanisms underlying Time-Warping are ported to operate in a distributed fashion, and are executed by the nodes that host the partially replicated data, when processing incoming transactional read and write operations. Then, when an update transaction requests to commit, a consensus procedure is executed to conduct the commit — this is a Two-Phase Commit (2PC) in the case of SCORe, but we could use other alternatives such as an Atomic Broadcast that delivers commit requests in total-order to the processes where they executed [Ruivo et al., 2011].

In the commit procedure each involved node executes read- and write-set validations, including those of Time-Warp, in an individual fashion. As such, each node will check whether the transaction is locally known to be a source or a target of an anti-dependency. Furthermore, it finds the minimum serialization point to which it may time-warp (in case it is the source of some anti-dependency). The semi-visible readers scheme is also applied at this point for update transactions, whereas read-only transactions perform it immediately during the read operation, similarly to what we did with TWM.

Given all of this, when each node concludes the validation, it decides on the fate of the

Table 4.6: Description of data-structures used in the DTW algorithms. The underlined fields are additions of DTW with respect to the original SCORe.

type	field	description
Transaction	id	unique identifier for the transaction
	ro	whether the transaction is read-only
	ts^S	timestamp for the snapshot visible to read operations
	ts^C	timestamp reflecting the commit order of the transaction
	$\underline{ts^W}$	timestamp reflecting the serialization order of the transaction
	mustTW	whether the transaction must time-warp to commit
	cannotTW	whether the transaction cannot time-warp to commit
	writeSet	the set of keys and values written by the transaction
	readSet	the set of keys and timestamps read by the transaction
Key	versions	set of committed versions for the key
	readStamp	timestamp for the last time the key was read
Version	ts^C	timestamp that reflects the commit order of the transaction that committed this version of the key
	$\underline{ts^W}$	timestamp that reflects the serialization order of the transaction that committed this version of the key
	timeWarped	whether it was committed by a time-warped transaction

transaction locally — that is, whether it should commit or abort — and sends its vote to the node that is coordinating the transaction (i.e., the one that initiated it). Then, the coordinating node makes the final decision by checking whether, among all the nodes involved, there were identified both source and target anti-dependences (in which case it decides to abort). Else, if any source anti-dependency has been identified, it performs a time-warp commit to the minimum serialization timestamp among all those that were sent back by the nodes.

As hinted above, for what concerns Time-Warping, its applicability in a distributed setting is quite straightforward. Most of the changes are adaptations of the TW mechanisms to the settings of partially replicated DTM. A transaction T accesses data residing at independent (distributed) nodes, which are later responsible for validating different subsets of the data items accessed by T. Hence, the validation outcome for T has to be determined at the coordinator node, after having gathered information from those various nodes replicating the data it accessed.

In the following we present the pseudo code for the implementation of DTW on top of SCORe, for which we describe the data-structures used for the metadata in Table 4.6. We also resort to some functions to improve the readability of the code:

Algorithm 11: Distributed Time-Warping pseudo code for accessing data.

```
1:  BEGIN(Transaction tx, bool ro) in node_i = origin(tx)
2|    tx.mustTW ← tx.cannotTW ← false
3|    tx.ro ← ro                                          ▷ optimize for read-only txs
4|    tx.id ← getUniqueId()
5|    tx.ts^S ← node_i.lastCommit                        ▷ obtain the snapshot of visible versions
6|    tx.ts^W ← ⊥                                        ▷ initialize special value: no time-warp

7:  WRITE(Transaction tx, Key k, Value v) in node_i = origin(tx)
8:    tx.writeSet ← tx.writeSet ∪ ⟨k, v⟩                ▷ defer write to commit-time

9:  READ(Transaction tx, Key k) in node_i = origin(tx)
10:   if k ∈ tx.writeSet
11:     return tx.writeSet.get(tx)                      ▷ read-after-write case: return deferred write
12:   send READREQ[k, tx] to all n_j ∈ owners(k)
13:   wait READREPLY[val] from any n_j ∈ owners(k)
14:   return val

15: upon receive READREQ[k, tx] in node_j ∈ owners(k)
16|   if tx.ro then updateReadStamp(tx, k)              ▷ make the read access visible
17:   acquireLock(k, SHARED)
18:   val ← localRead_SCORe(k, tx)                      ▷ delegate the local read to SCORe
19:   releaseLock(k)
20:   reply READREPLY[val]

21| updateReadStamp(Transaction tx, Key k) in node_i ∈ owners(k)
22|   atomically do {
23|     ⟨ stamp, id ⟩ ← k.readStamp
24|     newStamp ← node_i.lastCommit                    ▷ make read visible at present time
25|     if newStamp > stamp
26|       k.readStamp ← ⟨ newStamp, tx.id ⟩             ▷ update timestamp for one tx
27|     else k.readStamp ← ⟨ stamp, φ ⟩                 ▷ φ = several readers
28|   }
```

- *owners*(k) returns the nodes that replicate item k.
- *origin*(T) returns the node where transaction T originates from.
- *participants*(T) returns the nodes that replicate items written or read by transaction T.
- *local*(k) returns whether the data item k is replicated at the local node.

Then, in Algorithm 11 we present the functions to deal with the beginning of a transaction and read/write operations that it performs during its execution. The steps to execute are analogous to those of TWM, in the sense that writes are deferred and reads are performed with possibly updating the read timestamp — which happens for read-only transactions, as those skip the commit procedure given that they are always valid.

Algorithm 12: Distributed Time-Warping pseudo code for managing the commit of a transaction.

```
    ▷ commit procedure: join votes of the participants and decide
31: commit(Transaction tx) in node_i = origin(tx)
32:   send PREPARE[tx] to all n_j ∈ participants(tx)
33:   for all n_j ∈ participants(tx)
34:     wait VOTE[ _, vote_j] from n_j                    ▷ _ may be YES or NO
35|     tx.ts^W ← min(vote_j.ts^W, tx.ts^W)               ▷ possibly time-warp to the past
36|     if vote_j.mustTW
37|       tx.mustTW ← true
38|     if vote_j.cannotTW
39|       tx.cannotTW ← true
40:   if (∃ VOTE[NO, vote_j]) ∨ (tx.mustTW ∧ tx.cannotTW)
41:     send ABORT[tx] to all n_j ∈ participants(tx)
42:     return ABORT
43:   tx.ts^C ← max(votes.ts^C)          ▷ use the "most" recent timestamp known (SCORe rule)
44|   if not tx.mustTW then                              ▷ if tx does not have to time-warp
45|     tx.ts^W = tx.ts^C                                 ▷ then tx serializes at the present
46:   send COMMIT[tx] to all n_j ∈ participants(tx)

47: upon receive PREPARE[tx] in node_i ∈ participants(tx)
48:   for all k ∈ tx.writeSet : local(k)
49:     acquireLock(k, EXCLUSIVE)
50|     ⟨ stamp, id ⟩ ← k.readStamp
51|     ▷ check if any concurrent transaction B read data item k
52|     if stamp ≥ tx.ts^S ∧ id ≠ tx.id then              ▷ if tx does not have to time-warp
53|       tx.cannotTW ← true                  ▷ the concurrent B missed this tx (B --→ tx)
54:   for all ⟨k, ts⟩ ∈ tx.readSet : local(k)
55:     acquireLock(k, SHARED)
56:     ▷ check concurrently installed versions missed by tx (i.e., ∃A : tx --→ A)
57|     for all V ∈ k.versions : V.ts^C > tx.ts^S ∧ V.ts ≠ ts
58|       if V.timeWarped ∨ k ∈ tx.writeSet then        ▷ if tx does not have to time-warp
59|         reply VOTE[NO, tx]                  ▷ completes a triad; may not be serializable
60|         return
61|       tx.mustTW ← true                    ▷ tx missed some A (i.e., ∃A : tx --→ A)
62|       tx.ts^W ← min(tx.ts^W, V.ts^C)       ▷ compute time-warp timestamp for tx
63|     updateReadStamp(tx, k)                 ▷ update txs make reads visible at commit-time
64:   tx.ts^C ← fetchAndInc(node_i.nextId)      ▷ increment logical present time
65:   reply VOTE[YES, tx]

66: upon receive COMMIT[tx] in node_i ∈ participants(tx)
67:   atomically do {
68:     finalize_SCORe(tx)          ▷eventually invokes writeBack function described next
69:   }

    ▷ invoked for each write of tx when it is ready to commit
70: writeBack(Transaction tx, Key k) in node_i : local(k)
71|   newVersion.ts ← tx.ts^W                ▷version the write with the serialization timestamp
72|   newVersion.ts^C ← tx.ts^C
73|   newVersion.timeWarped ← tx.mustTW
74|   k.prependNewVersion(newVersion)          ▷newer versions are in the head of the list
```

In Algorithm 12 we present the functions to manage a commit request for an update transaction. This follows the protocol overviewed above, in which the coordinator of the transaction requests each participant (i.e., each node replicating data that the transaction accessed) to issue a validation and corresponding vote for the transaction. The logic of this validation follows closely that of time-warping. Finally, for successfully committed transactions, their writes are placed at the head of the list of the versions of each transactional datum[3].

4.9.2 Evaluation for Distributed Time-Warping

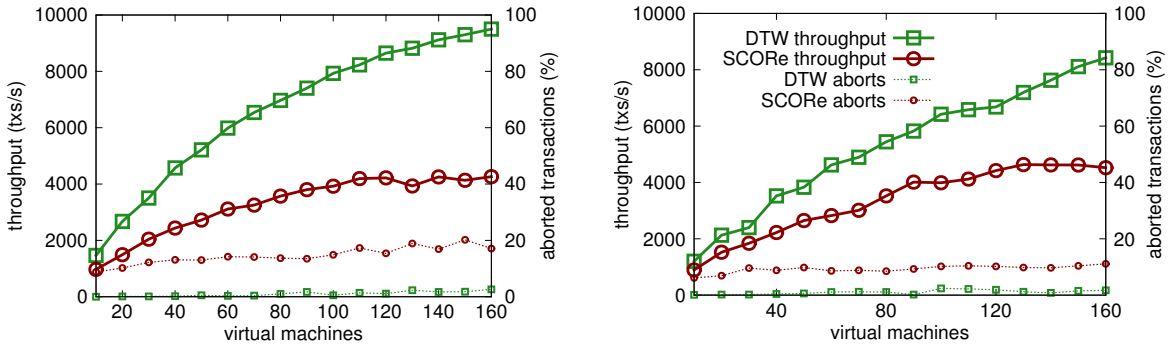
We conducted experiments on top of OpenStack, a cloud computing infrastructure, deployed in a dedicated cluster of 20 physical machines. Each such machine is equipped with two 2.13 GHz Quad-Core Intel Xeon E5506 processors, 40 GB of RAM, and is interconnected via a private Gigabit Ethernet.

We then deployed DTW and SCORe as Java processes on Virtual Machines (VMs) on top of this infrastructure via OpenStack. Each VM was provided with 1 physical core and 4 GB RAM. This represents a common scenario of deployment in cloud infrastructures, where customers acquire several virtual machines equipped with relatively small physical resources. For all tests we varied the number of VMs from 10 to 160, such that they were always uniformly distributed across the physical machines. As such we allocate up to 8 VMs per machine, allowing 8 GB of RAM left to the host operating system. Finally, the virtualized operating system was Ubuntu 12.04 and our prototype ran on Java HotSpot version 1.6.0_38.

We measure both overall throughput and abort probability (note that read-only transactions do not abort). Every run uses replication degree of 2 for fault-tolerance — hence data is considered durable once a transaction is committed. We then compare the performance of DTW with the DTM that it extends, namely, SCORe.

In Figure 4.18 we show the results for two benchmarks that we ported from the previous evaluation to a distributed setting, namely the Skip List and Vacation. The idea is that they are still written in a similar way, with accesses to shared data encapsulated in atomic blocks, but now the data resides in distributed machines and is partially replicated for fault-tolerance.

[3]Note that, in this implementation, we assume no blind-writes (i.e., a transaction always issues a read to datum k before writing to k). This has the effect of simplifying the write back procedure as there cannot exist a situation in which a more version exists without having created a triad (in contrast with what we had implemented in TWM, where blind-writes are allowed).



(a) Distributed Skip List.

(b) Distributed Vacation.

Figure 4.18: Performance and abort ratio of DTW compared to its baseline DTM.

As expected, the reduction of conflicts achieved by DTW is analogous to that of our initial proposal of Time-Warping with TWM. In Skip List, with a workload of 50% read-only transactions, we achieve a peak improvement of 2.23× over SCORe when using the maximum number of VMs, thanks to a consistent reduction of aborts for the update transactions from an average of 15% to 1%. All of this is achieved by DTW without sacrificing the genuineness property, which is evident in the scalability unfolded by it. With Vacation we witness a similar scenario, although the gains are smaller due to the lower amount of aborts that SCORe yields: 1.9× performance improvement with 160 VMs.

4.10 Summary

In this chapter we proposed the idea of Time-Warping, with its implementation in TWM, a novel multi-version algorithm that aims at striking a balance between permissiveness and efficiency. TWM exploits the key idea of allowing update transactions to be serialized "in the past", according to what we called a *time-warp* time line. This time line diverges from the natural commit order of transactions in order to allow update transactions to commit successfully (but in the past) despite having performed stale reads. Past solutions have tried to avoid spurious aborts by introducing costly and inefficient transaction validation procedures. TWM explored a new validation strategy that results in less aborts, without hindering efficiency (e.g., by avoiding expensive checks of the transactions' dependency graph). Furthermore, TWM ensures mv-permissiveness (abort-free read-only transactions) and VWC (a strong and practical correctness criterion).

We conducted a broad experimental study aimed to empirically quantify: (1) what abort

reduction rate can be achieved by TWM, and (2) whether TWM is sufficiently lightweight to actually benefit from the abort reduction that it can yield. The study encompassed a variety of benchmarks, and alternative TM algorithms, optimized to minimize either bookkeeping overheads or spurious aborts. Our experimental data evidenced the merits of time-warping, which achieved an average improvement of 65% in high concurrency scenarios and gains extending up to 9×. Further, we showed that TWM introduces very limited overheads when faced with contention patterns that cannot be optimized using TWM.

Finally, we also extended the idea of Time-Warping to distributed settings, namely to reduce the conflicts generated by the implementations of DTMs. The ease of applicability of this idea to that setting improves also the significance of this work.

5 Tuner: Self-tuning the HTM Fall-back

As highlighted by our study in Chapter 3, the best-effort nature of HTMs (namely, of Intel RTM) can have a profound impact on performance. The limitations of current HTMs are quite pragmatic, greatly motivated by the reliance on the processors' hardware caches, which have limited space, to track transactional accesses.

Indeed, although solutions providing stronger progress guarantees for HTM have been proposed in literature, the changes required to existing processor architectures are currently perceived as overly invasive and risky [Adir et al., 2014][1], and it appears unlikely that alternative designs will be pursued in the near term future.

With such HTMs, which require software to manage their usage, programmers need to set up retry policies for when to use hardware transactions or an alternative synchronization mechanism (the fall-back path). Choosing these policies can not only be cumbersome, contradicting the simplicity advocated in the TM paradigm, but it may also lead to sub-optimal situations in which the programmer chose a retry policy that is not adequate to all possible workloads of the application. This is, indeed, one of the results that we obtained in the initial study of the current state-of-the-art in TM in Chapter 3.

In this chapter we shall address this issue via a self-tuning algorithm that manages the usage of the HTM with respect to an alternative synchronization mechanism (we used both a single-global lock and an STM). To achieve this, our novel algorithm — called Tuner — uses lightweight reinforcement learning techniques that seek the optimal configuration to use. Similarly to the approach taken in the previous Chapter 4, this allows to preserve the simple TM abstraction and avoids imposing any burden on the programmer.

[1] IBM System Z processors represent a notable exception: they guarantee transactions to commit if they abide certain constraints in terms of footprint and instructions, provided that they do not conflict. However, we note that those requirements are quite strict (at most 32 instructions, accessing up to 256 bytes of memory).

5.1 Problem

Using such best-effort HTMs, however, entails providing a software fall-back path such as those that we studied in Chapter 3 in co-operation with Intel RTM. As such, a programmer who is responsible for a TM library using hardware transactions in RTM must decide what should be done upon the abort of that hardware transaction: under which circumstances should an aborted transaction be re-executed using RTM, or when should it resort to an alternative fall-back software-based synchronization scheme?

In this chapter we show that there is no definite answer to this question: indeed, there is no one-size fits all solution that yields the best performance across all possible workloads. In fact, this is something that we had also conveyed when conducting the comparison of synchronization techniques, namely in Section 3.6, which we seek to improve with this dissertation to obtain better performance robustness for TM.

To better support this important claim, we provide experimental evidence summarized in Figure 5.1. This figure shows the performance of three example configurations, using Intel RTM, across several TM benchmarks. A detailed description of these configurations will be provided in Section 5.5. These results show the relative performance of each configuration, with respect to the optimal one that we found for each benchmark. The outcome of this data supports our claim: no configuration performs consistently better than all the others, and they all perform excellently in some benchmarks and poorly in others.

This important fact means that the programmer is left with the responsibility of finding out the best choice of configuration for his application - a problem that is cumbersome and time consuming to tackle via off-line profiling, given that there are many available configurations. Even worse, in fact, there may not exist a single optimal solution to be found statically, in the cases where dynamic workloads are present. These facts have also been recently acknowledged by Intel researchers, highlighting the importance of developing adaptive techniques to simplify the tuning of RTM [Kleen, 2014].

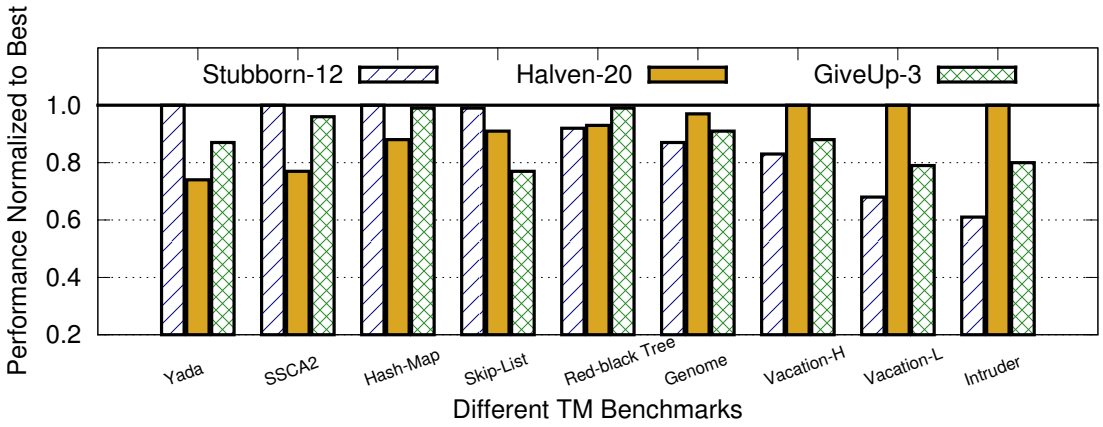


Figure 5.1: Relative performance of three example RTM configurations with respect to the best static configuration in each benchmark with 8 threads (we show 9 different benchmarks). The configurations used differ in the number of retries allowed for hardware transactions and in how they deal with different types of aborts. This experimental data highlights that no configuration performs consistently better than the others, and that all static configurations can be far from the optimal performance at least for certain workloads.

5.2 Overview

With this work we contribute to addressing the aforementioned challenge by studying the problem of automatically tuning the policies used to regulate the activation of the fall-back path of RTM. In more detail, our contributions are structured as follows:

- In Section 5.5, we show that self-tuning is essential to achieve robust performance across different workloads. In particular, we present evidence showing that no single configuration exists that outperforms all others. Furthermore, we find that the performance of any static configuration can deliver losses up to 10× when compared to the optimal solution for each individual workload.

- In Section 5.6 we present a novel solution that relies on lightweight reinforcement learning techniques to perform runtime adaptation based on the online monitoring of applications' performance. For this to be possible, it is crucial to reduce any overhead of the profiling and to strike a good balance between exploring new configurations and exploiting available knowledge on already sampled configurations. In Section 5.7 we also discuss different designs for our solution, and their inherent trade-offs. We then explain, in Section 5.8, how our algorithms have been integrated in the GCC compiler in order to achieve full

transparency for programmers. This is an important contribution as we preserve the simple abstraction of TM, and effectively conceal our self-tuning mechanisms for adapting the TM configuration given the current workload characteristics.

- Finally, we instantiate our TM adaptive approach considering two distinct RTM fall-back paths: namely with a fall-back based on a single-global lock, as well as an STM-based fall-back (i.e., a HyTM). We study their performance, in Section 5.10, by using a large set of TM benchmarks, in which we obtain average gains of 60% over the best static alternatives and up to two-fold improvements in specific benchmarks. This is obtained also while achieving a performance gap, with respect to the optimal configuration in each benchmark, that is $< 5\%$.

5.3 Related Work

Works that studied the performance of RTM [Yoo et al., 2013, Karnagel et al., 2014] — including that in Chapter 3 — have so far used static configurations, which were found to have the best average performance for the considered workloads after extensive manual explorations. Similar studies were performed for IBM processors, also lacking workload-oblivious and application-independent mechanisms for tuning the HTM usage [Jacobi et al., 2012, Su and Heisig, 2013, Odaira et al., 2014]. However, as highlighted in the previous sections, and discussed in more detail in Section 5.5, the choice of the optimal configuration is strongly workload dependent.

Another recent work, called Adaptive Lock Elision (ALE) [Dice et al., 2014b], proposed to regulate the choice of when to switch from HTM to software fall-back paths, by relying on an ad-hoc adaptive approach that samples different configurations and interpolates the estimated performance of those not tested. As such, a fundamental difference between our approaches is that we explore the whole space of configurations, via reinforcement learning techniques that aim to identify a sweet-spot between exploration and exploitation.

Additionally, there have been other proposals for self-tuning STMs. Adaptive Locks [Usui et al., 2009], VOTM [Leung et al., 2013], and Dynamic Pessimism [Sonmez et al., 2009] adapt between optimistic (with STM) and pessimistic (via locking) execution of atomic blocks. Unfortunately, these approaches are not tailored to the specific problem studied in this chapter, and

as such the resulting performance is disappointing as we show in our evaluation (by considering Adaptive Locks, as the authors kindly provided us with their code, and whose high-level approach is similar to the other mentioned works). More complex adaptation schemes have been proposed to self-tune the choice between different STM algorithms in AutoTM [Wang et al., 2012b]. The main drawback of this kind of works, with regard to the HTM setting studied in this dissertation, is that these self-tuning proposals require knowledge that is not available from the HTM support that we have, such as the footprint of transactions (their read- and write-sets). That is, unless we instrument reads and writes to obtain it, which would defeat the purpose of HTM in avoiding the instrumentation overheads that are inherent to STMs.

We summarize the comparison with the related work in Table 5.1. There, we highlight the target of each self-tuning algorithm and the approach taken in each work.

Finally, there is also a large body of work that has focused on adapting orthogonal aspects of TM algorithms, namely: the ideal mapping of locks to shared memory addresses [Felber et al., 2008]; the optimal parallelism degree, in terms of application threads, in both STM [Didona et al., 2013] and HTMs [Rughetti et al., 2014, Mohamedin et al., 2015a]; and searching for the optimal mapping of threads to processor cores, so that they benefit the most from the cache usage and avoid mutual interference with one another [Castro et al., 2014]. In contrast with these works, we focused on dealing with the emerging challenges posed by best-effort HTMs, such as Intel RTM, i.e., self-tuning the policies that govern the retry logic and the usage of the software fall-back path in best effort HTMs.

5.4 Preliminaries

In the course of this chapter we focus on self-tuning RTM when coupled with a single-global lock or alternatively the NOrec STM. We have presented and studied these two approaches in Chapter 3, under the name of, respectively, RTM-GL and RTM-NOrec.

In both cases we extend the base Algorithm 1 that was initially introduced to explain how RTM can be used to execute atomic blocks in co-operation with a software fall-back path (a global lock in that specific case). We augment that base algorithm in all our approaches — baselines included — to avoid the *lemming effect* [Dice et al., 2008], by having the thread wait for the fall-back path not to be under use, before every attempt to start the transaction.

Table 5.1: Comparison of self-tuning algorithms that aim to increase the performance of TM systems by automatically choosing some parameters. Our proposal, Tuner, is the only one targeting best-effort HTMs and using a simple and lightweight approach that requires no effort of re-writing the applications.

Self-tuning algorithm	Tuning Parameter	Granularity	Approach	Inputs
ALE [Dice et al., 2014b]	between HTM, Optimistic and Pessimistic Locking	runtime atomic block	blind exploration with ad-hoc extrapolation scheme	throughput, conflicts and time used
Adaptive Locks [Usui et al., 2009]	between STM and elided lock	source-code atomic block	analytical model	throughput in each mode
VOTM [Leung et al., 2013]	number of threads allowed in each atomic block concurrently	source-code atomic block	analytical model	cycles spent in committed vs aborted transactions
DynLocks [Sonmez et al., 2009]	optimistic versus pessimistic locking	transactional variable	thresholds based on heuristics	number of aborts
AutoTM [Wang et al., 2012b]	STM algorithm	whole application	machine learning and heuristics	transaction length, conflicts, throughput, and other features
Tuner	HTM retry policy configuration	source-code atomic block	online exploration with reinforcement learning	cycles spent in each configuration

As stated in the overview of the contributions, we propose to integrate our improvements in the TM implementation that is used by GCC, which is applied by the compiler to programs written with atomic blocks. As such, we use the base GCC algorithm as one of the baselines to motivate our work, as its performance is highly affected by the static configuration that is hard-coded in its source-code. Its RTM usage is very similar to our base Algorithm 1, with two differences: hardware transactions are only used as long as the status returned by RTM has the `retry` bit set (recall Table 2.2), and the wait to avoid the lemming effect occurs only if the transaction has already aborted once.

We also point out that we use RTM-GL as the driving example for the presentation of this chapter, due to its simplicity when compared to RTM-NOrec. Still, the core ideas are analogous, and we consider both approaches in the full evaluation study in Section 5.10.

Finally, all the evaluation conducted in this chapter uses the machine previously described in Table 3.2 (and used in our comparative study in Chapter 3), which, we recall, has 4 physical cores and 8 hardware threads with support for Intel RTM.

5.5 Making the Case For Adaptation: No One-size Fits All

The software-based transaction retry logic, regardless of being backed by a single-global lock or STM fall-back, can be governed via two main tuning knobs: 1) the budget (i.e., the total number) of attempts available for hardware transactions, before resorting to the fall-back path, and 2) how to consume such budget depending on the causes of transactions' aborts.

In order to assess the performance impact that these configuration options can have in practice, we conducted an experimental study in which we considered a configuration's space containing all possible combinations of feasible values of the following two parameters:

- Budget of attempts — Varying from 0 to 20. This means that the software handler may choose to avoid at all to use RTM, or to insist up to 20 times before stop calling it to execute a transaction successfully.
- Capacity aborts — From our early Table 2.2, we can see that it is possible to obtain feedback from RTM about capacity overflow aborts. In such cases we allow three possible configurations: GiveUp exhausts all attempts upon a capacity abort; Halven drops half

of the attempts on a capacity abort; and STUBBORN decreases one attempt only. The objective is to have different behaviours according to how serious the impediments are to commit successfully in hardware [2].

We tested all the possible combinations of the above configuration space in our suite of benchmarks, with varying concurrency degrees, reaching the conclusion that there is no *one-size fits all* solution. In Table 5.2 we show some of the results from these experiments. For the sake of brevity, we focus on the scenario of 4 threads.

We show the speedups (of the whole benchmark duration, relative to a non-instrumented sequential execution) of:

- The best configuration found via exhaustive off-line search.
- A policy which we call GIVEUP-5 (see Algorithm 13).
- And of the TM algorithm used by the GCC compiler, which we name GCC, as described in the previous Section 5.4.

The GIVEUP-5 policy is inspired by recent works, which have used a static value of 5 attempts in their configuration [Yoo et al., 2013, Karnagel et al., 2014]. Also, as the name suggests, this policy resorts to the fall-back path upon the first occurrence of a capacity abort. We consider this configuration in our study in order to include a static policy that takes reasonable choices regarding the setting of the retry budget and the management of capacity exceptions.

In general, the GIVEUP-5 algorithm yields some considerable improvements over GCC. However, the most important results are on the rightmost column, where we can see that the best performing variant varies significantly in its characteristics. For instance, in the SSCA2 benchmark there are very few aborts due to the limitations of RTM, so the best configuration is stubborn with respect to capacity aborts and uses up to 10 hardware transaction retries. In contrast, Yada does trigger many deterministic capacity aborts, so it is best to desist immediately upon their occurrence; however, if the transaction aborts for other reasons, it is actually better to retry 7 times.

[2] We focus on capacity aborts because this is the only specific reason placed in the EAX register that may represent a deterministic impediment for a hardware transaction to succeed. In our experience, abort reasons captured by events in Model Specific Registers (MSRs) are too expensive to monitor online in a fine-grained fashion because reading them implies issuing system calls.

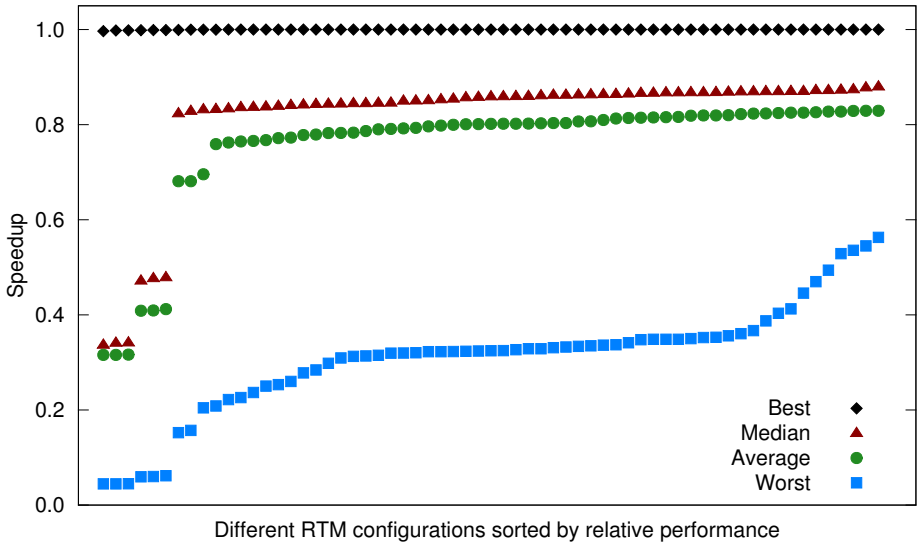


Figure 5.2: Speedups — across all benchmarks and number of threads evaluated in this chapter — of different RTM configurations with respect to the optimal configuration for the considered experiment. For each data point we consider four statistical metrics over the speedups: the best, median, average and worst speedups that each configuration obtained across all the experiments (benchmarks and number of threads).

Overall we can see that the optimal configuration varies significantly across different benchmarks. The complexity of identifying a single static configuration emerges even more clearly when considering Figure 5.2, which reports speedups across all benchmarks considered (several data-structures and the STAMP benchmarks, described further in Section 5.10) and number of the threads (from 1 to 8). Each speedup point corresponds to the performance of some RTM configuration, relative to the optimal configuration for the considered experiment (identified off-line via an exhaustive exploration). The tested configurations result from the cross product of

Table 5.2: Speedup of static configurations with 4 threads relatively to a sequential execution in some of the benchmarks used in our evaluation.

Benchmark	GCC	GiveUp-5	Best	Configuration
Genome	0.65	2.64	2.84	Halven-9
Intruder	0.73	2.48	3.05	Stubborn-4
Kmeans-high	2.74	2.85	2.99	Halven-3
Labyrinth	0.99	0.99	1.00	GiveUp-9
SSCA2	2.81	2.88	3.27	Stubborn-10
Vacation-high	0.74	1.76	2.64	Stubborn-19
Yada	0.79	0.87	0.92	GiveUp-7

the three aforementioned ways to deal with capacity aborts and by varying the number of retries from 1 to 20 (described in the previous Section 5.5).

By analyzing the experimental data reported in Figure 5.2, it is possible to see that the performance loss with respect to the optimum for *any* configuration can range from 0.4× to 10×. Furthermore, all configurations performed quasi-optimally in some benchmark and parallelism degree, as evidenced by the data points of the Best line shown. Overall, this experimental data, together with the additional data reported in Sections 5.1 and 5.10, provide strong arguments on the lack of a one-size-fits-all configuration and represent a compelling motivation for the proposed self-tuning schemes.

The bottom line of this section is that static configurations of RTM can deliver suboptimal performance as a consequence of the high heterogeneity of the workloads generated by TM applications. The experiments above illustrate how much one could gain in the considered set of benchmarks, workloads and concurrency degree, by using an approach that adapted dynamically the RTM configuration to the workloads' characteristics. Naturally, it is undesirable to require the programmer to identify an optimal configuration for each workload, in particular because workloads may be unpredictable or even vary over time. In fact, we highlight that, in principle, adaptive approaches may yield even results better than the single, static best performing configuration for a given benchmark, e.g., in case its workload changes over the course of execution (as it is the case in Intruder), or because different atomic blocks have very different characteristics (as in Labyrinth).

5.6 Self-tuning Intel Restricted Transactional Memory

Our approach to tackle the problem of "no one-size fits all" in the RTM software fall-back is to perform online lightweight profiling and adaptation. We use a simple self-tuning feedback loop based on a target performance metric and instantiate reinforcement learning algorithms that guide the exploration of the possible configurations towards the optimal one. This allows to better fit the workloads of typical irregular applications that benefit most from synchronization with TM [Dragojević et al., 2011], for which fully offline optimizations are likely to fall prey of the over-approximations of solutions based on static analysis techniques. Another appealing characteristic of the proposed approach is that it does not necessitate any preliminary training

Algorithm 13: Possible static configuration of RTM (we call it GiveUp-5).

```
1:  HTM_START()
2:    attempts ← 5                          ▷ increased budget of attempts to sustain spurious aborts
3:  begin:
4:    while(is-locked(sgl)) do x86_pause                          ▷ avoid the lemming effect
5:    htmStatus ← _xbegin()
6:    if htmStatus ≠ _XBEGIN_STARTED
7:      if attempts = 0
8:        acquire-lock(sgl)
9:        return
10:     else
11:       if htmStatus = capacity
12:         attempts ← 0                    ▷ give up, likely to always fail due to capacity overflow
13:         goto line 8
14:       else
15:         attempts ← attempts - 1
16:       goto begin
17:   if is-locked(sgl)
18:     _xabort()
19:   return

20: HTM_END()
21:   if _xtest()
22:     _xend()
23:   else
24:     release-lock(sgl)
```

phase, unlike other self-tuning mechanisms for Software TM (STM) based on off-line machine-learning techniques [Rughetti et al., 2012, Didona et al., 2014].

Clearly, keeping the overhead of our techniques very low is a crucial requirement, as otherwise any gain is easily shadowed, for instance due to profiling inefficiencies or repeated decision-making. Another challenge is the constant trade-off between exploring alternative configurations versus exploiting the current one, with the risk of getting stuck in a possibly sub-optimal configuration.

The proposed solution seeks to minimize overhead in a twofold way. First, it avoids any synchronization among concurrent threads, and it employs simple metrics for performance sampling — such as x86's Time Stamp Counter (TSC) cycle counter. Besides that, it employs a combination of lightweight techniques, borrowed from the literature on reinforcement learning and hill climbing algorithms, which were selected, over more complex techniques, precisely because of their high efficiency.

Another noteworthy feature of the proposed self-tuning mechanism is that it allows for

individually tuning the configuration parameters of *each* application's atomic block, rather than using a single global configuration. This feature is particularly relevant in programs that include transactions with heterogeneous characteristics (e.g., large vs small working sets, are contention-prone or not, etc.), which could benefit from using radically different configurations.

Before detailing the proposed solution, we first overview a state of the art solution [Auer et al., 2002] for a classical reinforcement learning problem, the multi-armed bandit [Sutton and Barto, 1998]. This reinforcement learning technique is the key building block of the mechanism that we use to adapt the policy used to deal with capacity aborts, which will be described in Section 5.6.2. We then explain the adaptation of how stubborn should one be in using RTM, i.e., the budget of attempts, in Section 5.6.3. The combination of those techniques is presented in Section 5.6.4.

5.6.1 Bandit Problem and UCB

The "bandit" problem (also known as "multi-armed bandit") is a classic reinforcement learning problem that states that a gambling agent is faced with a bandit (a slot machine) with K independent arms, each associated with an unknown reward distribution. The gambler iteratively plays one arm per round and observes the associated reward, adapting its strategy to maximize the average reward. Formally, each arm i ($0 \leq i < K$) — where K is the number of arms — is associated with a sequence of random variables $X_{i,n}$ representing the reward of the arm i, where n is the number of times the lever has been used. The goal is to learn which arm i maximizes the average reward (μ_i) computed as:

$$\mu_i = \sum_{n=1}^{\infty} \frac{1}{n} X_{i,n} \tag{5.1}$$

To this purpose, the learning algorithm needs to try different arms to estimate their average reward. On the other hand, each suboptimal choice of an arm i costs, on average, $\mu^* - \mu_i$, where μ^* is the average obtained by the optimal lever. Several algorithms have been studied to minimize this regret (i.e., loss with respect to the optimal), defined as:

$$\mu^* n - \mu_i \sum_{i=1}^{K} E[n_i] \tag{5.2}$$

where n_i is the number of times arm i has been chosen, given that there are K total arms.

Building on the idea of confidence bounds, the technique of Upper Confidence Bounds (UCB) creates an overestimation of the reward of each possible decision, and lowers it as more samples are drawn. Implementing the principle of optimism in the face of uncertainty, the algorithm picks the option with the highest current bound. Interestingly, this allows UCB to achieve a logarithmic bound on the regret value not only asymptotically, but also for any finite sequence of trials. In more detail, UCB assumes that rewards are distributed in the [0,1] interval, and associates each arm i with a value:

$$\bar{\mu}_i = \bar{x}_i + \sqrt{2\frac{\log n}{n_i}} \tag{5.3}$$

where $\bar{\mu}_i$ is the current estimated reward for lever i; n is the number of the current trial; $\bar{x}_i$ is the reward for lever i; and n_i is the number of times the lever i has been tried. The right-hand part of the sum is an upper confidence bound that decreases as more information is acquired. By choosing, at any time, the option with maximum $\bar{\mu}_i$, the algorithm searches for the option with the highest reward, while minimizing the regret along the way.

As such, the intuition behind UCB is to balance the trade-off between exploring new choices versus exploiting those already known. The advantage of UCB is in providing bounds on the errors committed over time when searching for the optimal choice.

5.6.2 Using UCB learning

We applied UCB by considering that each atomic block of the application is associated with a bandit, i.e., a corresponding UCB instance. With it, we seek to optimize the choice of what to do when a capacity abort occurs in a hardware transaction. In some sense, this models a belief on whether the capacity aborts witnessed are transient or deterministic, a fact that cannot be established correctly based only on the error codes returned by aborted transactions.

We tackle the issue of how to manage capacity aborts by gathering feedback on the performance yielded when using the three options identified in Section 5.5, i.e., 1) decrementing linearly the number of retries, 2) halving the number retries, and 3) using the fall-back path upon the occurrence of the first capacity abort. We associate each option with a bandit lever and use the UCB algorithm to solve the trade-off between exploration and exploitation, i.e.,

testing strategies that appear to be sub-optimal based on available evidence, versus selecting the strategy that, so far, yielded on average the best performance.

The rationale that motivates the choice of using only these three alternative strategies is to keep the number of UCB levers low. This is a relevant design, as the higher the number of bandit's levers, the longer time UCB will spend exploring sub-optimal solutions.

In order to instantiate Equation (5.3), we associate with n the number of decisions taken so far for the atomic block, and with n_i the number of times the UCB instance chose lever i. As for the reward function associated with the levers (represented by $\bar{x}_i$), we consider the number of processor cycles that it takes to execute (i.e., commit, after possibly some retries) the atomic block using each different lever/strategy. To this end, we associate a counter c_i with each lever that we use to track how many cycles were consumed so far, i.e., to execute n_i times the atomic block using lever i. We also keep track of the best (i.e., lowest) number of cycles so far to process atomic block i as $best_i$. Then, we compute the reward $\bar{x}_i$ for lever i with:

$$\bar{x}_i = \frac{best_i}{c_i/n_i} \tag{5.4}$$

which means that we normalize the cycles of lever i, giving us a reward in the interval [0,1]. If a given configuration leads to executing an atomic block fast enough (i.e., close to the best number of cycles spent so far) then this produces a high reward. In contrast, for slow executions, the average number of cycles in the denominator will be larger than the best recorded value on the numerator, and thus the reward will be low.

5.6.3 Using Hill Climbing Exploration

The other space of configurations that we want to optimize is the number of retries to use in hardware transactions before triggering the fall-back software path. The optimization problem that we want to solve here is illustrated by the experiments in Figure 5.3, where we show the performance improvement of using RTM with 8 threads relative to a sequential version of the code (with no instrumentation), in the STAMP benchmark Vacation, when varying the number of attempts for the configuration. In the plot we show two workloads for Vacation that generate low and high contention, for which there are significantly different number of attempts yielding maximum values of improvement (namely, 12 and 16).

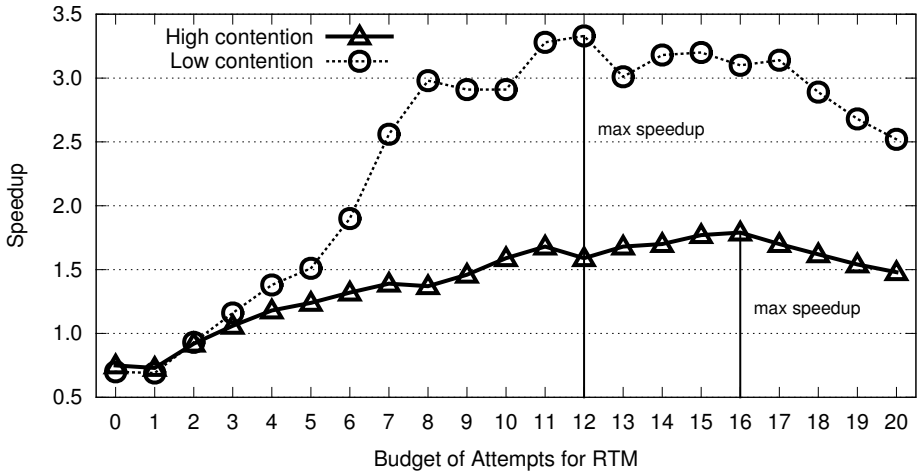


Figure 5.3: Speedup in Vacation when varying budget of attempts for RTM usage, 8 threads, and two contrasting workloads varying contention to shared memory (using the GiveUp strategy to deal with capacity aborts).

In order to optimize the number of attempts configured for each atomic block, we use an exploration technique, similar to hill climbing search [Russell and Norvig, 2009]. Its idea is to perform an iterative search, in which each step consists of evaluating some gain function: if the last change (i.e., increment or decrement by one unit) of the budget of retries produced an improvement, then the same change is applied again.

The alternative of using UCB was dismissed because the parameter — number of attempts — has a large space of search that makes UCB quite cumbersome (e.g., UCB requires an initial, uniform sampling of *every* possible configuration).

To implement the function that measures the gain we use also the processor cycles that it takes to execute the atomic block. We augmented the classical hill climbing with probabilistic jumps to avoid getting trapped in local maxima during the exploration. This is because these classical techniques end up exploring neighbourly configurations and may not be able to escape a local maxima that is surrounded by worse configurations — even though a global optimal configuration may exist elsewhere. Furthermore, we memorize the best configuration seen so far to recover from unfortunate probabilistic jumps that yield excessive performance degradation.

To implement all of this, we store in our library: the best configuration and performance seen so far (*best*); the last configuration and corresponding performance (*last*); and the current configuration (*current*). Note that the configuration means simply the number of attempts.

Then we use the following rules to guide exploration (strategy called HC):

1. With probability 1-p_{jump} play according to classic hill climbing; if performance improved along the current direction of exploration, keep exploring along that direction; otherwise reverse the direction of exploration.

2. With p_{jump} probability, select randomly the attempts with uniform probability for the next configuration. If after the jump performance decreased by more than $maxLoss$, then revert to the *best* known configuration. As we shall see in the next section, when merging the usage of HC (to configure the number of retries) with that of UCB (to configure the reaction to capacity aborts), we actually create a rule to decide better the extent and direction of these probabilistic jumps.

As we shall see, in the implementation of these techniques in Section 5.8, we perform this search/optimization step with some periodicity of a number of transactions. That is, we do not explore a new configuration on every new transaction but, instead, let several (e.g., a hundred) transactions execute before triggering a new exploration.

Further, in order to enhance stability and avoid useless oscillations once identified the optimal solution, if, after a configuration change, performance did not change by more than $min\Delta$, we block the hill climbing exploration and allow only probabilistic jumps (to minimize the risk of getting stuck in sub-optimal configurations).

Concerning the settings of the p_{jump}, $maxLoss$, and $min\Delta$, we set them respectively to 1%, 40% and 5%, which are typical values for this type of algorithms [Sutton and Barto, 1998] and whose appropriateness in the considered context will be assessed in Section 5.10.

5.6.4 Merging the Learning Techniques

So far we have presented: 1) UCB to optimize the consumption of attempts upon capacity aborts (Section 5.6.2); and 2) HC to optimize the allocation of the budget of attempts (Section 5.6.3). We now present their integration in our algorithm called TUNER.

The concern with the integration in TUNER is that the two optimization strategies overlap partially in their responsibilities. The advantage is that this allows to simultaneously optimize the configuration accurately for atomic blocks that sometimes exceed capacity in a deterministic

way, whereas, in other scenarios, can execute efficiently using RTM. This may be, for instance, dependent on the current state of shared memory, or some input parameter used in the atomic block. It is possible to achieve this because UCB shall decide to short-cut the attempts when capacity aborts happen, whereas HC can keep the attempts' budget high to allow successful RTM execution when capacity aborts are rare.

One problematic scenario arises when an atomic block is not suitable for execution in hardware: either HC can reduce the attempts to 0, or UCB can choose the GIVEUP mode. However, we may be unlucky and get an inter-play of the two optimizers such that they affect each other and prevent convergence of the decisions.

To solve this problem with their integration, we create a hierarchy among the two optimizers, in which UCB can force HC to explore in some direction and avoid ping-pong optimizations between the two. For this, we create a rule that is activated when the attempts' budget is exhausted: in such event we trigger a random jump to force HC to explore in the direction that is most suitable according to UCB, that is, explore more attempts if the UCB belief is STUBBORN and less attempts otherwise.

We compute the extension of the random jump for HC (based on the direction decided by UCB), by taking into account information about the types of aborts incurred so far. Namely, we collect the number of aborts due to capacity (*ab-cap*) and due to other reasons (*ab-other*). Then, if UCB suggests exploring more attempts (i.e., UCB belief is STUBBBORN), we choose the length of the jump, noted J, proportionally to the relative frequency of *ab-other*:

$$J = \frac{ab\text{-}other}{ab\text{-}cap + ab\text{-}other} \cdot (maxTries - cur)$$

where *cur* is the current configuration of the budget of attempts and $maxTries = 20$. If UCB is different from STUBBORN, the jump has negative direction, and length:

$$J = \frac{ab\text{-}cap}{ab\text{-}cap + ab\text{-}other} \cdot cur$$

We now assess the efficiency of each of the optimization techniques alone, and their joint approach described above as TUNER. In this joint strategy we seek to understand if the two optimization techniques work well together: Figure 5.4 shows the speedup of TUNER relatively to UCB and HC individually, with each individual strategy using the respective static configuration

from GiveUp-5 (i.e., UCB uses 5 attempts and HC uses GiveUp) — we average the results across benchmarks since they yielded consistent results.

We can see from our experiments that the joint strategy provided average results that are always better than at least one of the approaches alone. More than that, for most cases TUNER improved over both individual strategies, which shows that employing them in synergy provides better results than the best approach alone. This is an encouraging result because tuning the attempts and dealing with capacity aborts is not entirely a disjoint concern. Overall, the results show that the joint approach yielded up to 20% improvement. Notice that each technique individually already improves over the baselines presented earlier, so any improvement when merged further reduces the gap with respect to the optimal result.

5.7 Granularity of Tuning

We now consider the trade-offs in possible designs of our proposal for self-tuning RTM.

On one extreme, it is possible to self-tune RTM for each atomic block defined in the application. This fits better applications with high heterogeneity of atomic blocks: it is conceivable to imagine that one atomic block a_1 defined by the programmer may execute very fast and with a small footprint, whereas another one a_2 in the same application is very large and often exceeds the capacity of RTM. As such, a_1 may be optimized to have a high budget of attempts and not to insist upon capacity aborts, whereas a_2 may immediately give up RTM when faced with a capacity abort.

On the other extreme, we may self-tune the application as a whole, independently of its atomic blocks. In contrast with the alternative above, this has the advantage of requiring less

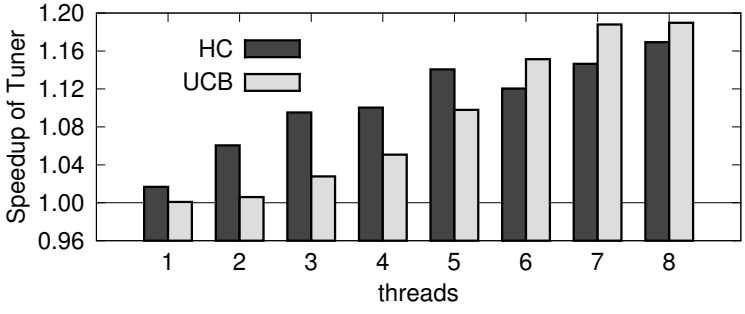


Figure 5.4: Geometric mean speedup of TUNER over UCB and HC across benchmarks and threads.

metadata: a single UCB and hill climbing instances, instead of one per atomic block. As such, it also takes less executions of each atomic block to gather statistically meaningful data to conduct the reinforcement learning procedures, as they are all gathered into the same metadata.

By avoiding multiple concurrent optimizations, this design has the benefit of avoiding possible interference. As an example, atomic block a_1 may be experimenting with a strategy that causes it to give up and acquire the global lock often (in the case of RTM-SGL). As a result, now atomic block a_2 — often executed concurrently with a_1 — gets aborted in hardware frequently due to the lock being taken by a_1, and this may mislead the optimizer used by a_2. Due to the interference of a_1's optimizer, in fact, a_2's optimizer may associate a low reward with the strategy, say s, currently in use for a_2. However, if a_1 eventually were to converge to a different strategy, strategy s may eventually become optimal for a_2 and have, at steady state, a higher reward. Indeed, by performing self-tuning at the granularity of the whole application (and not of the individual atomic blocks), we avoid this sort of interferences, although at the cost of a reduced tuning flexibility.

Finally, it is also possible to perform the optimizations on a per-thread basis, or globally across threads. This raises similar trade-offs to those above, as multiple concurrent optimizations may have undesirable side-effects that mislead the other reinforcement learners. Conversely, a single global optimization across all threads may fail to capture heterogeneity in the threads: it is conceivable to imagine an application in which some threads execute atomic block a_1 with different parameters that cause them to have heterogeneous access patterns for the same code in a_1. As such, different threads may have very different behaviors within a_1, for which different RTM usages are optimal.

To capture the different trade-offs of these designs, we create two versions of our proposal. We call the algorithm that optimizes each atomic block independently in a per-thread fashion TUNER, which . In contrast, we call the global version of the algorithm G-TUNER, as it optimizes the whole application (considering all atomic blocks as one) and with all threads using the same learner and following the same optimization.

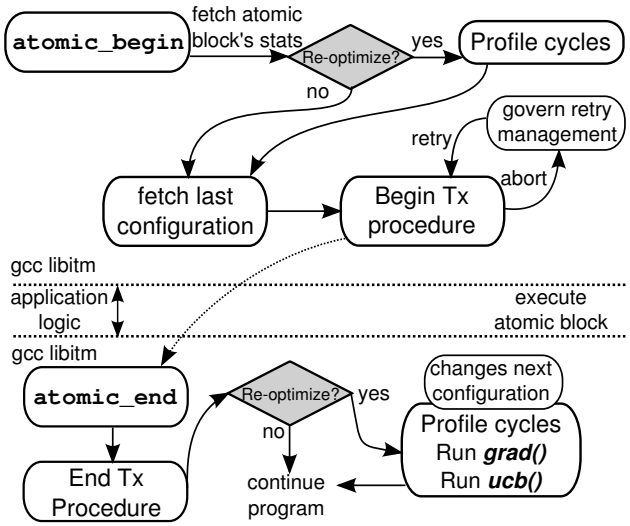


Figure 5.5: Workload-Oblivious tuning of RTM.

5.8 Implementation Details

In this section we provide additional details on our implementations. We focus on the case of RTM-SGL, for its simplicity, although the principles that we explain are the same for the application of the proposed tuning strategies to RTM-NOrec. We begin by explaining how we integrated our algorithms in the latest stable version of the Gnu C Compiler (GCC version 4.8.2), inside its *libitm* component. This component is responsible for implementing the C++ TM Specification [Adl-Tabatabai et al., 2012] in GCC that allows programmers to write atomic constructs in their programs, which are compiled to contain calls to our TM runtime library. Therefore, this allows programmers to benefit from the improvements of our proposal fully transparently.

We now consider our TUNER, which uses per atomic block statistics and configurations. For this, we modified GCC to uniquely identify each atomic block, and to pass that information to calls to the TM runtime. We begin by laying out a high-level description of TUNER in Figure 5.5. To the purpose of this presentation, G-TUNER shares most of its implementation with TUNER, thus, despite their antagonistic design choices, we describe mostly how to integrate TUNER. We then describe the details that differ among the two.

The flow in Figure 5.5 starts every time a thread enters an atomic block, at which point the corresponding metadata of the atomic block is fetched, by using its unique id. Every per atomic block metadata is maintained in thread-local variables: hence threads perform self-tuning in an independent fashion. This has the advantage of avoiding synchronization and allowing threads

Algorithm 14: TUNER applied to RTM-SGL.

```
 1: HTM_START()
 2:   ucbBelief ← ▷ retrieve from the last configuration used
 3:   attempts ← ▷ retrieve from the last configuration used
 4:   if reoptimize()
 5:     initCycles ← obtainRDTSC()
 6: begin:
 7:   while is-locked(sgl) do x86_pause
 8:   htmStatus ← _xbegin()
 9:   if htmStatus ≠ _XBEGIN_STARTED
10:     if attempts = 0
11:       if reoptimize() then tuneAttempts(ucbBelief)
12:       acquire-lock(sgl)
13:       return
14:     else
15:       if htmStatus = capacity
16:         ▷ set the attempts of the next configuration according to ucbBelief; rules of Section 5.6.4
17:       else
18:         attempts ← attempts - 1
19:       if attempts = 0
20:         goto line 11
21:       goto begin
22:   if is-locked(sgl)
23:     _xabort()
24:   return

25: HTM_END()
26:   if _xtest()
27:     _xend()
28:   else
29:     release-lock(sgl)
30:   if reoptimize()
31:     totalCycles ← obtainRDTSC() - initCycles
32:     ucbBelief ← UCB(totalCycles)        ▷ rules of Section 5.6.2
33:     attempts ← HC(totalCycles)          ▷ rules of Section 5.6.3
```

to reach different configurations, which can be useful in case the various application threads are specialized to process different tasks (and generate different workloads).

After fetching the metadata, we check whether it is time to re-optimize the configuration for that atomic block. This condition is a result of the sampling that we use to profile the application. For this, we keep a counter of executions in the metadata of the atomic block (recall that it is thread local) so that we only re-optimize periodically — we set this period to 100 transactions in our evaluation. This classic technique allows to keep the overheads low without missing noticeable accuracy in the decisions taken [Leung et al., 2013, Usui et al., 2009, Wang et al., 2012b]. Hence we place the check for re-optimization in the begin and end of the atomic block. In the negative case, we simply execute the atomic block with the last configuration set

up for it and proceed without any extra logic or profiling.

In the case that we re-optimize, this enables profiling of the cycles that it takes to execute the atomic block. For this, we use the `RDTSC` instruction in x86, which we use as a lightweight profiling tool to measure the relative cost of executing the block in different configurations. After this we attempt to start the transaction itself, which is better described in Algorithm 14. Lines 10-21 describe the retry management policy. During a re-optimization period, if the attempts' budget is exhausted, this triggers the forced random jump over HC according to the description of TUNER in Section 5.6.4 (line 11), before proceeding to the fall-back path. Note also that upon a capacity abort we adequately reduce the available budget according to the belief of UCB set in the current configuration (line 16).

After the application executed the atomic block, it calls back to *libitm*, and TUNER executes the usual procedure to finish the transaction. After this, it checks whether it is re-optimizing the atomic block, and in the positive case it runs HC and UCB to adapt the configuration for the next executions. To do so, it uses the processor cycles consumed, and applies the rules described throughout Sections 5.6.2 and 5.6.3 to configure the budget and consumption of attempts in the metadata of the atomic block.

Similarly to other metrics, assessing performance via processor cycles is also subject to thread preemption, which may inflate the actual cost of executing the atomic block. We mitigate this issue by binding threads to logical cores [3], and evaluating scenarios with up to as many threads as logical cores, as more than those typically deteriorates performance anyway.

To conclude this section, we highlight how G-TUNER differs from the above description of the implementation of TUNER. In this case, we did not require the unique numbering of each atomic block, given that G-TUNER does not distinguish between atomic blocks, and instead optimizes the application as a whole. For this, every thread keeps the metadata of the executed atomic blocks, commits and the different number and type of aborts that it witnessed, in per-thread variables. An additional adapter thread, spawned by our runtime in *libitm*, periodically collects that profiled data and conducts the optimizations of UCB and HC. As such, it is no longer responsibility of the application threads to do that procedure when they begin and end

[3]We note that in our evaluation setup, which has 8 hardware threads, we have confirmed that not binding threads to cores does not produce a statistically measurable difference in performance. This, however, may not remain true in machines with architectures different from the one used in this work.

and atomic block, as we explained for TUNER. The resulting configuration, obtained by the adapter thread, is published in a global structure, which every application thread queries before executing atomic blocks. This leads to all threads to follow the same optimized strategy.

5.9 Opportunities for Optimizations and Extensions

We now discuss possible extensions to our work that would optimize it in different scenarios that are not addressed in the TM benchmarks that are typically used in the literature and in our following evaluation. That is to say, while they should improve performance in some scenarios, that would not be evident in the set of benchmarks on which we perform the evaluation.

5.9.1 Granularity of the Self-tuning

In this work we presented self-tuning proposals that adapt the choice of the RTM fall-back for a given (or all) atomic blocks presented in the source-code. This works best in the case each atomic block is often invoked from the same context, i.e., its call stack and relevant scoped variables are the same.

Let us present a contrasting example to explain why that is the case. Suppose that a concurrent red-black tree is written with an atomic block around an insert operation (as is the case in our evaluation following up). Then, an application may instance two such red-black trees in the same piece of code and then populates them with contrasting workloads: one tree may be very small and the other very large. As a result, the same insert operation will now be invoked over these different trees, and the resulting workload will be dramatically different as one will typically run fine in RTM whereas the other may fail deterministically due to capacity aborts.

While this limitation motivates for a different granularity for the self-tuning — for instance, atomic blocks that consider also the call-stack — we note that this kind of behaviour does not occur in a measurable way on the reference benchmarks for TM systems that are used in the literature (and in our evaluation). Namely, it is often the case that an atomic block is invoked from the same code location, generating identical workloads in subsequent invocations. Even if the workload changes, as is the case for some STAMP benchmarks, this happens gradually and not with concurrent inter-leaved requests exhibiting different workload characteristics as was the case for the example given above.

A possible solution to address this issue is to use the approach of ALE proposed by Dice et al. [Dice et al., 2014b], which consists of labeling atomic blocks with context identifiers that capture invocations from different contexts in the code. This allows a given static atomic block — i.e., at the source code level — to be seen as different instances at runtime. The integration of this technique with TUNER is perfectly viable, and would enable different optimizations of the same atomic block depending on the context in which it is invoked.

5.9.2 Bootstrapping the Self-Tuning Process

When presenting TUNER we argued for the use of thread-local variables to maintain the metadata necessary for the self-tuning process. This approach has two main advantages. First, it allows threads to perform self-tuning in an independent fashion. This can be beneficial for applications in which threads generate heterogeneous workloads and, consequently, have different optimal configurations. Second, our distributed self-tuning approach is designed to avoid any inter-thread synchronization.

However, for applications that often spawn new threads, this poses the problem that such threads basically restart the learning process by collecting statistics from scratch. For such cases, which are not encompassed in our evaluation test-bed, we propose a simple idea borrowed from G-TUNER: we can periodically gather statistics across the threads into a centralized metadata, and a new thread may use that to bootstrap its self-tuning.

5.9.3 Workload Changes

The self-tuning approach taken here is theoretically tailored to stationary workloads. This is because the UCB solution, which we employed to decide what to do upon capacity aborts, assumes a constant reward function. In fact, some of the STAMP benchmarks that we evaluate our solution with, have dynamic workloads that change over time. As we shall see, our solution performs efficiently (compared to any static solution) even in those cases.

Regardless, it is worth highlighting other alternatives that have the potential to perform as good (or even better) with more suitable theoretical guarantees. One alternative would be to use a different solution to the bandit problem that accounts for possible changes in the workload [Kaufmann et al., 2012, Garivier and Moulines, 2011]. The idea there is to consider the

data collected on the levers of the bandit over a sliding window of time. Alternatively, we could use a workload change detector, such as CUSUM [Basseville and Nikiforov, 1993], and reset the UCB statistics upon a workload change.

5.10 Evaluation Study

We now present our final experimental study, in which we assess the effectiveness of our self-tuning proposal Tuner. In addition to RTM-SGL, which we have discussed extensively in this chapter, we consider also the HyTM RTM-NOrec that we used in Chapter 3 (that is, the Hybrid NOrec of Section 2.5.1 with several optimizations and Reduced Hardware Transactions). We enhance each of these two approaches with both Tuner and G-Tuner, as described in the previous Sections 5.5 and 5.8.

We note that we have only implemented the RTM-SGL approaches (both for Tuner and G-Tuner) in GCC. For approaches with RTM-NOrec, instead, we resorted to a macro-based library that the applications invoke directly. The main reason being the lack of support in libitm for HyTMs as they require two instrumentation paths — besides the non-instrumented one — whereas libitm provides support only for one instrumented path. We highlight that our RTM-SGL implementations yield equivalent performance both when used in GCC (by optimizing for the non-instrumented path when possible) and also in a macro-based library. As such, we believe that the same would be apply for RTM-NOrec, should libitm be optimized also for HyTMs.

We compare to the baselines that were described throughout this dissertation, together with a state of the art approach that was designed in the scope of STMs:

- GCC: as described in Section 5.4, corresponding to the implementation available in *libitm* in GCC 4.8.2.
- AdaptiveLocks: was proposed to decide between locks and TM for atomic blocks [Usui et al., 2009]; an analytical model is used and fed with statistics sampled at runtime (similarly to Tuner). We adapted their code (using CIL) to our environment integrated in GCC.
- GiveUp-5: corresponding to Algorithm 13, which embodies the best practices described in the literature to statically tune RTM with heuristics. We applied this baseline to both

RTM-SGL and RTM-NOrec.

- STUBBORN-10: unlike GIVEUP-5, this heuristic insists on using RTM for 10 times independently of the exception type that caused the transaction to abort. Then, it switches to using the fall-back path. Also in this case we consider two fall-back paths, namely SGL and NOrec.

- Best Static: an idealized upper bound on the best result possible, obtained by picking the best configuration among all those possible for each benchmark and degree of parallelism. As such, this alternative does not correspond to a real tuning algorithm, but rather to an optimal, static configuration specifically tailored for each workload/benchmark. We show the results using this ideal variant both for the case of RTM-SGL and RTM-NOrec.

Similarly to all the results shown already, we shall present speedup values that are computed with respect to a sequential, non-synchronized execution of each benchmark. Every experiment was repeated ten times to achieve statistically meaningful results.

We begin by summarizing our findings across all benchmarks in the following Section 5.10.1. Then, in Section 5.10.2, we study the performance of the different solutions in a set of concurrent data-structures, which are widely used to study TM algorithms. Afterwards, in Section 5.10.3, we extend our study to a variety of popular TM standard applications. Finally, we also evaluate the algorithms from an energy-efficiency perspective in Section 5.10.4.

5.10.1 Summary of Evaluation

To ease the interpretation of all our results in this extensive study, we begin by summarizing our findings across all the benchmarks that we use (namely, concurrent data-structures in Section 5.10.2 and the STAMP benchmarks in Section 5.10.3), whose detailed description is provided later in the following sections.

First of all, we sought to understand the overhead of our self-tuning approaches. For that, we created variants of our TUNER and G-TUNER (applied to both RTM-SGL and RTM-NOrec) that executed all the profiling and optimization procedures, even though the atomic blocks were always executed with a static configuration. We compared the performance of these stripped down tuning algorithms with a baseline that used that same static configuration and had none

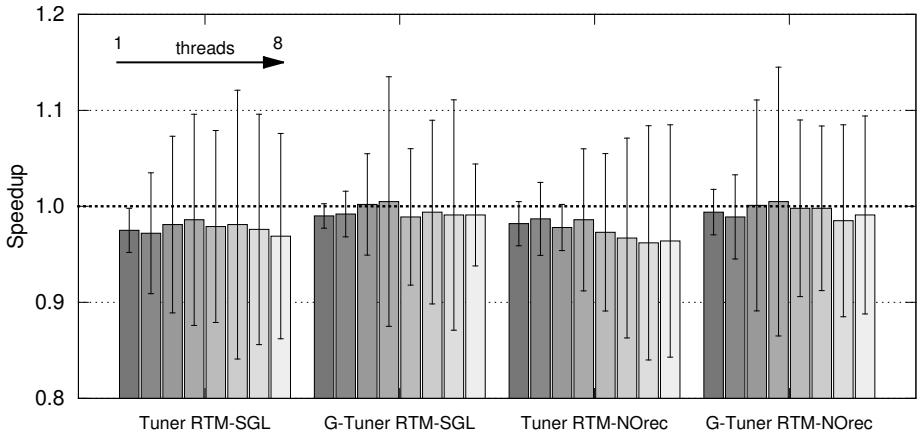


Figure 5.6: Overhead of the self-tuning algorithms. For each algorithm, we used a stripped down version in which all the profiling and optimizations are performed, but the RTM configuration used is always the same. We compare the performance with a baseline that uses that same static configuration and show the geometric mean speedup across all benchmarks (data-structures and STAMP).

of the added procedures. The results, which illustrate the overheads of our proposed solution, are shown in Figure 5.6.

This experiment shows that the overhead of G-Tuner is slightly less than Tuner, although in any case the values amount at most to an average of roughly 5%. This is an inspiringly low value, which we hope to capitalize with the self-tuning produced by these algorithms.

We then present the average speedup across all benchmarks, for each approach, relative to that of Best Static. As such, values closer to 1 are optimal. The results for RTM-SGL and RTM-NOrec are presented, respectively in Tables 5.3 and 5.4. The main result that we highlight here is two-fold: 1) there is a large gap between the performance of the baselines and that of the Best Static variant, regardless of the used fall-back path; and 2) both Tuner and G-Tuner are capable of effectively minimizing that gap, yielding an average performance that is very close to that of the Best Static variant.

Note that the worst performing approaches (namely, GCC and AdaptiveLocks) tend to have higher standard deviation, because in some benchmarks they perform closer to the optimal (e.g., Labyrinth and SSCA2) and in others they can perform much worse (e.g., Vacation and Intruder). In contrast, the best performing approaches tend to perform more consistently throughout the benchmarks.

Table 5.3: Speedup of each approach in **RTM-SGL** relative to the Best Static (with standard deviation) averaged across all benchmarks.

Speedup to Best Static	threads			
	2	4	6	8
GCC	0.61 ± 0.31	0.39 ± 0.34	0.31 ± 0.33	0.25 ± 0.34
AdaptiveLocks	0.67 ± 0.25	0.54 ± 0.29	0.45 ± 0.31	0.40 ± 0.32
GiveUp-5	0.90 ± 0.13	0.86 ± 0.11	0.81 ± 0.18	0.76 ± 0.24
Stubborn-10	0.88 ± 0.12	0.83 ± 0.12	0.80 ± 0.16	0.74 ± 0.21
Tuner	0.95 ± 0.05	0.95 ± 0.07	0.97 ± 0.07	0.97 ± 0.12
G-Tuner	0.96 ± 0.04	0.95 ± 0.06	0.96 ± 0.06	0.96 ± 0.10

Table 5.4: Speedup of each approach in **RTM-NOrec** relative to the Best Static (with standard deviation) averaged across all benchmarks.

Speedup to Best Static	threads			
	2	4	6	8
NOrec	0.53 ± 0.31	0.53 ± 0.30	0.55 ± 0.31	0.53 ± 0.32
GiveUp-5	0.85 ± 0.12	0.83 ± 0.13	0.76 ± 0.18	0.70 ± 0.21
Stubborn-10	0.81 ± 0.18	0.78 ± 0.19	0.70 ± 0.20	0.61 ± 0.24
Tuner	0.92 ± 0.05	0.93 ± 0.05	0.95 ± 0.10	0.95 ± 0.14
G-Tuner	0.94 ± 0.05	0.95 ± 0.04	0.96 ± 0.10	0.97 ± 0.12

In fact, for 8 threads, we can see that our self-tuning proposals achieve performance within 5% of the Best Static across both types of fall-backs. The data appears to suggest that Tuner performs slightly better in RTM-SGL and G-Tuner in RTM-NOrec. We argue that this may depend on the inherent trade-offs that exist between the two proposed solutions:

- In RTM-SGL, bad configurations for triggering the fall-back are more catastrophic, because dictating the usage of the fall-back imposes heavy serialization on the single-global lock. As such, it pays off more to have a finer-grained optimization with Tuner, even though the overhead of doing so is higher.

- In RTM-NOrec, the fall-back is less aggressive in constraining concurrency as it uses an STM rather than a global lock. As such, it better to have a more coarse-grained (and lighter) optimization, by relying on G-Tuner instead.

Although these antagonistic design choices pose a trade-off, the aggregated performance values are fairly close to each other. In the following sections we shall delve into detailed results for each benchmark, where the differences become noticeable. Furthermore, it is important

Table 5.5: Parameters used in the data-structures tested. The low (and respectively high) contended workloads across data-structures were chosen in a way such that they generate approximately the same degree of contention.

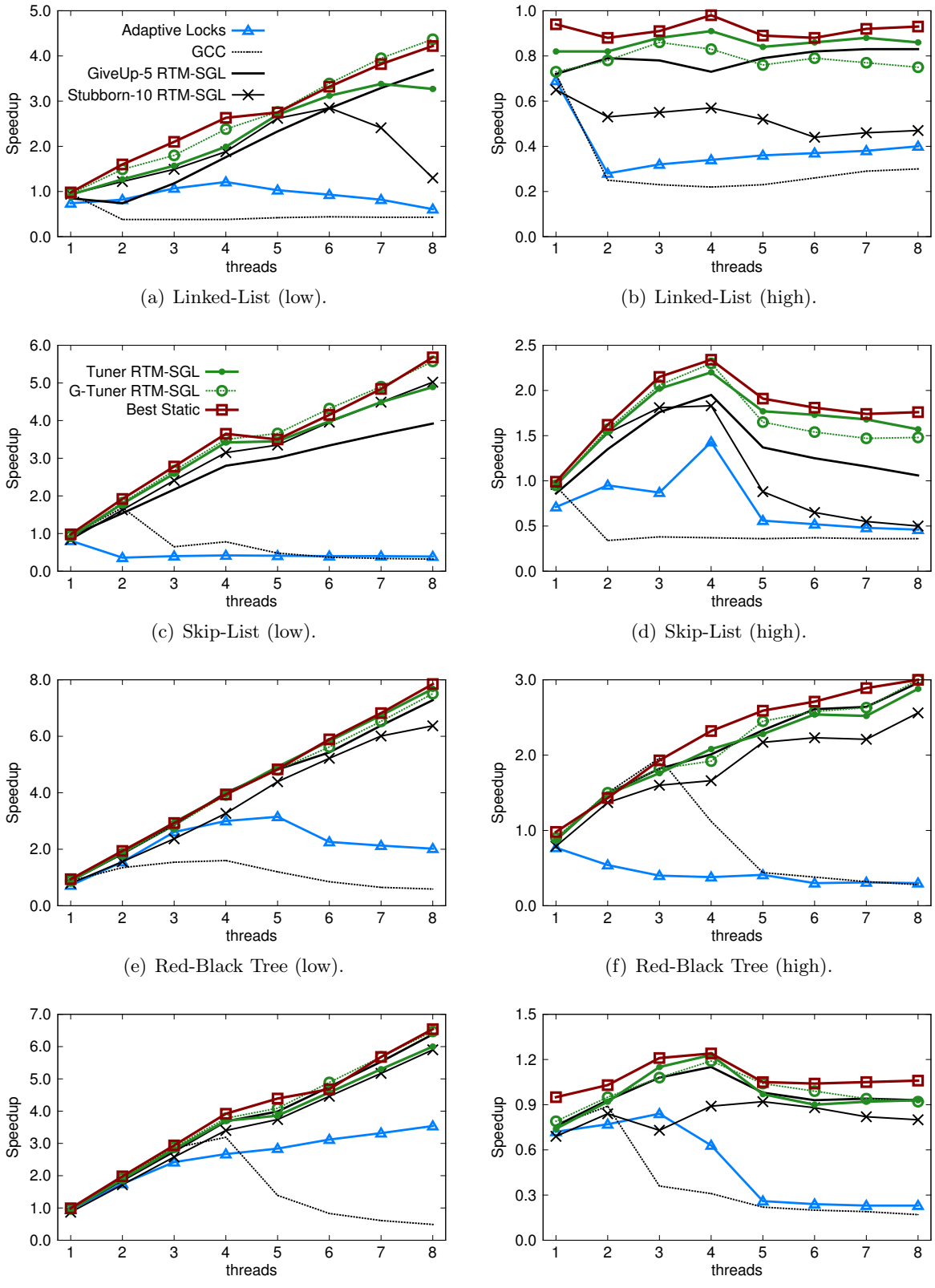
Benchmarks (contention)	Size	Insert / Remove / Contains
linked-list (low)	1000	1% / 1% / 98%
linked-list (high)	1000	37% / 37% / 26%
skip-list (low)	5000	1% / 1% / 98%
skip-list (high)	10000	45% / 45% / 10%
red-black tree (low)	100000	5% / 5% / 90%
red-black tree (high)	1000	45% / 45% / 10%
hash-map (low)	10000	5% / 5% / 90%
hash-map (high)	100	45% / 45% / 10%

to recall that TUNER allows for self-tuning workloads that are heterogeneous across threads, which does not happen in these typical TM benchmarks. We have verified this in synthesized benchmarks: for instance, one benchmark where some threads manipulate a low contended hash-map and others a contended linked-list; in such case TUNER obtained about 20% improvement over G-TUNER for 8 threads. However, for cases where this heterogeneity between threads is not expected, then the simpler algorithm for G-TUNER provides approximately the same performance, which makes it more appealing.

Finally, we do not specifically aim to adapt between the different fall-backs — namely between using both the SGL and NOrec — for which reason it is out of scope to compare directly their performance here. In fact, others have proposed and evaluated the trade-offs between such approaches in the scope of Hybrid TMs [Matveev and Shavit, 2013, Dalessandro et al., 2011, Riegel et al., 2011, Dice et al., 2014b].

5.10.2 Concurrent Data-Structures

We tested the set of four representative data-structures that we presented in Section 2.7, which are typically used to study the performance of TM algorithms [Dalessandro et al., 2010, Hammond et al., 2004, Felber et al., 2008, Dragojević et al., 2009a]. We used two workloads for each data-structure, respectively with low and high contention. The parameters used are described in Table 5.5, and were chosen such that the low (and respectively high) contention across data-structures generated approximately the same ratio of aborted transactions (less than 10% for low contended, and more than 50% for high contended).



(a) Linked-List (low).

(b) Linked-List (high).

(c) Skip-List (low).

(d) Skip-List (high).

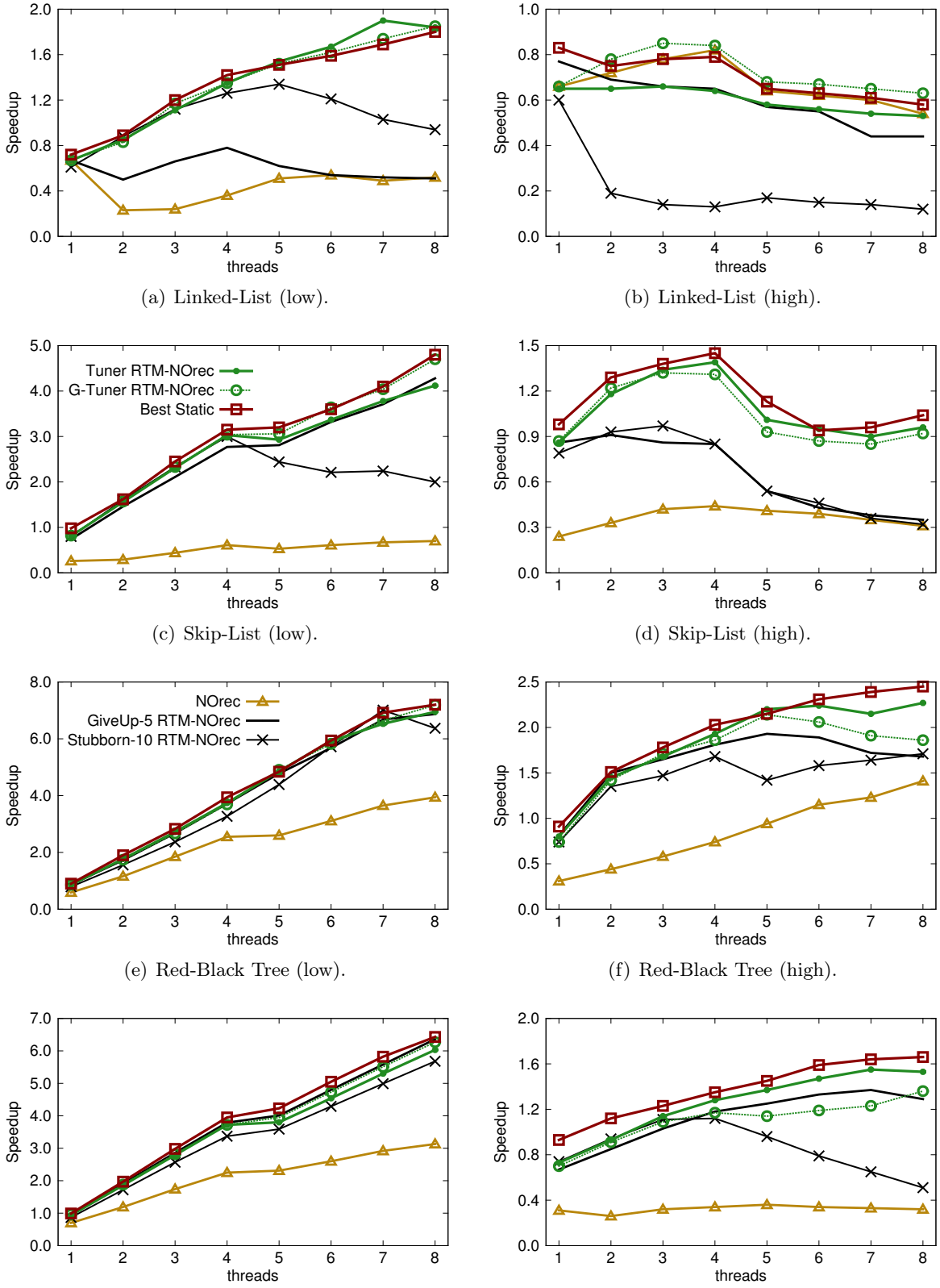
(e) Red-Black Tree (low).

(f) Red-Black Tree (high).

(g) Hash-Map (low).

(h) Hash-Map (high).

Figure 5.7: Speedups relative to sequential execution in the data-structures benchmarks when tuning **RTM-SGL** with different approaches.



(a) Linked-List (low).

(b) Linked-List (high).

(c) Skip-List (low).

(d) Skip-List (high).

(e) Red-Black Tree (low).

(f) Red-Black Tree (high).

(g) Hash-Map (low).

(h) Hash-Map (high).

Figure 5.8: Speedups relative to sequential execution in the data-structure benchmarks when tuning **RTM-NOrec** with different approaches.

We present our results concerning RTM-SGL in Figure 5.7 and RTM-NOrec in Figure 5.8. The highlight of most of our evaluation is that our self-tuning approaches are generally the closest ones to the idealized Best Static variant; not only that, but the gap between the optimal performance and our solution is typically very small.

In general, the static baselines can perform significantly worse than the Best variant. This is an expected result, as we have already shown how difficult it is for a static approach to perform optimally across workloads. The performance of Adaptive Locks, on the other hand, are largely unsatisfactory due to the large, constant, overheads that it entails. This approach, in fact, requires manipulating shared memory locations to store metadata associated with the atomic blocks. These manipulations require costly synchronization that hampers performance significantly and shadows the gains achievable via dynamic adaptation.

If we consider the two heuristics GiveUp-5 and Stubborn-10, we can see that they usually have contrasting results. That is, whenever one is closer to the Best Static, the other performs worse, and vice-versa. This is a consequence of their antagonistic heuristics for budget of attempts and to deal with capacity aborts.

If we compare our two tuning approaches, we can see across all the plots that they perform very similarly in general. There are some sporadic advantages towards G-Tuner that yield a geometric mean improvement of 5% over Tuner, which happens for two main reasons. Firstly, there are three atomic blocks only in these data-structures, one for each type of operation (and two of which are very similar, mutation operations). As such, the G-Tuner approach to optimize the application globally does not lose much with respect to the specialization of Tuner because there are few and homogeneous atomic blocks. Secondly, these atomic blocks are very small compared to those of some applications that we test later. Consequently, Tuner's main disadvantage, i.e., replicating the tuning procedure across different threads, becomes more noticeable. This is mainly a result of measuring the processor cycles spent, which adds a constant overhead of roughly 65 cycles in our machine, and is thus visible in such small and dominant atomic blocks.

While it is not our intention to compare the different RTM fall-backs, we can see between the two sets of plots that the scalability and performance trends are very similar. This is a result of the fact that the best performing algorithms allow both RTM approaches to rely on hardware transactions and, indeed, we have already assessed in Chapter 3 that the concurrent

Table 5.6: Configurations to which TUNER and G-TUNER converge when using **RTM-SGL** and running with 8 threads.

Benchmarks (contention)	RTM-SGL			
	Tuner			G-Tuner
	Insert	Remove	Contains	
linked-list (low)	GIVEUP-4	GIVEUP-2	GIVEUP-9	GIVEUP-8
linked-list (high)	GIVEUP-6	HALVEN-4	HALVEN-5	HALVEN-7
skip-list (low)	HALVEN-6	HALVEN-5	STUBBORN-5	HALVEN-8
skip-list (high)	GIVEUP-8	GIVEUP-6	HALVEN-15	GIVEUP-15
red-black tree (low)	HALVEN-7	HALVEN-7	STUBBORN-6	HALVEN-8
red-black tree (high)	GIVEUP-8	GIVEUP-8	STUBBORN-8	HALVEN-9
hash-map (low)	HALVEN-8	HALVEN-7	STUBBORN-9	HALVEN-9
hash-map (high)	GIVEUP-6	GIVEUP-6	HALVEN-5	GIVEUP-8

data-structures are very well suited for hardware transactions.

As the last set of experiments that use the data-structures, we also present the configurations for which our TUNER approaches converged to. This is a particularly interesting opportunity to delve into such details because these data-structures have stationary workloads and a small number of atomic blocks when comparing to the benchmarks evaluated later.

Because each thread in TUNER may choose a different configuration, we present the configuration that is chosen by most threads in a given run. In practice, given the homogeneous workload of these micro-benchmarks, we noticed that most configurations were similar across the threads. Furthermore, we present the configurations that were used for the longest time in the benchmark, as our self-tuning algorithms regularly re-optimize, and may change their decision over time. Finally, these results correspond to the case of running with 8 threads, as that is the case where the gains were generally larger with our self-tuning proposals.

The results are shown in Tables 5.6 and 5.7, respectively for RTM-SGL and RTM-NOrec. Note that, as explained above, we show three configurations for TUNER corresponding to the three atomic blocks used in the data-structures (the canonical insert, remove and contains operations). In the case of G-TUNER, a single configuration is chosen globally for the application.

Starting with RTM-SGL and TUNER, it is possible to identify several trends. First, the low contended workloads are generally more conservative when dealing with capacity aborts, by halving or remaining stubborn most of the time, when compared to their high contented counter-

Table 5.7: Configurations to which TUNER and G-TUNER converge to when using **RTM-NOrec** and running with 8 threads.

Benchmarks (contention)	RTM-NOrec			G-Tuner
	Tuner			
	Insert	Remove	Contains	
linked-list (low)	GIVEUP-9	GIVEUP-6	STUBBORN-10	STUBBORN-6
linked-list (high)	GIVEUP-6	GIVEUP-6	GIVEUP-15	GIVEUP-8
skip-list (low)	HALVEN-4	HALVEN-5	HALVEN-6	HALVEN-6
skip-list (high)	GIVEUP-1	GIVEUP-6	HALVEN-4	GIVEUP-16
red-black tree (low)	GIVEUP-6	HALVEN-6	STUBBORN-7	HALVEN-12
red-black tree (high)	GIVEUP-6	HALVEN-6	STUBBORN-5	GIVEUP-6
hash-map (low)	HALVEN-7	HALVEN-7	STUBBORN-8	STUBBORN-6
hash-map (high)	GIVEUP-16	HALVEN-7	STUBBORN-6	GIVEUP-12

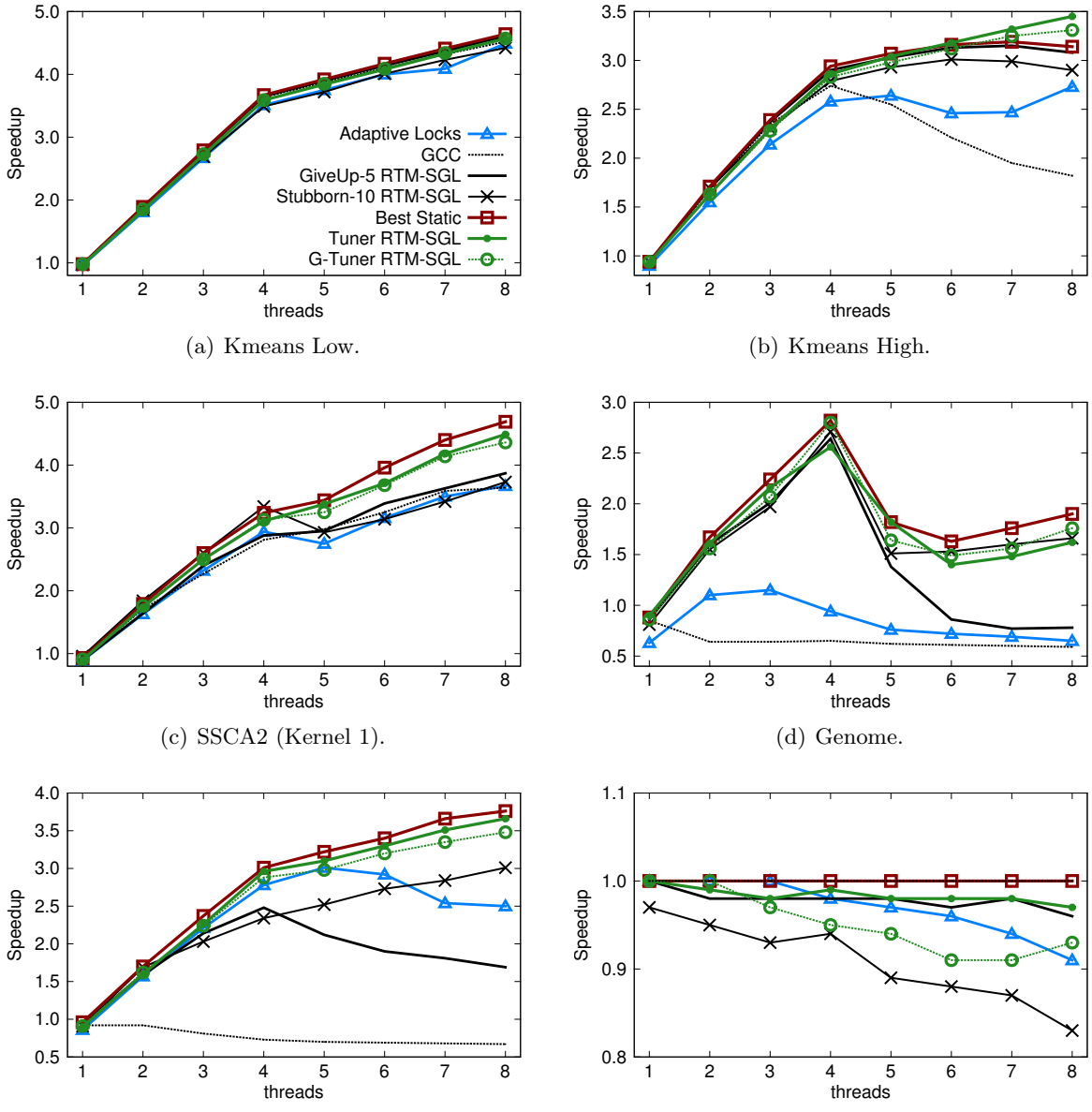
parts. We can also see that the Skip-List and Linked-List, which generate longer footprints for the transactions, lead TUNER to use a lower budget of attempts and to give up more frequently upon capacity aborts.

As for the configurations selected by G-TUNER, also in RTM-SGL, in most scenarios they appear to roughly approximate the *average* of the configurations identified by TUNER for each atomic block. Furthermore, also G-TUNER, in 3 of the 4 considered benchmarks, converges to more conservative policies for dealing with capacity aborts (i.e., it does not give up immediately) when faced with low contention workloads.

The data reported in Table 5.7, which refers to the RTM-NOrec case, exhibits analogous trends to the ones observed for the RTM-SGL case. If we compare each configuration individually, the trend here is that RTM-NOrec is more prone to giving up on RTM — i.e., after a lower number of attempts — which should be a result of the fall-back being much more efficient than the SGL used in the case of RTM-SGL.

5.10.3 Application Benchmarks

For representative application benchmarks we relied on the STAMP suite. We used the standard parameters for the STAMP benchmarks and show workloads for low and high contention when available.



(a) Kmeans Low.

(b) Kmeans High.

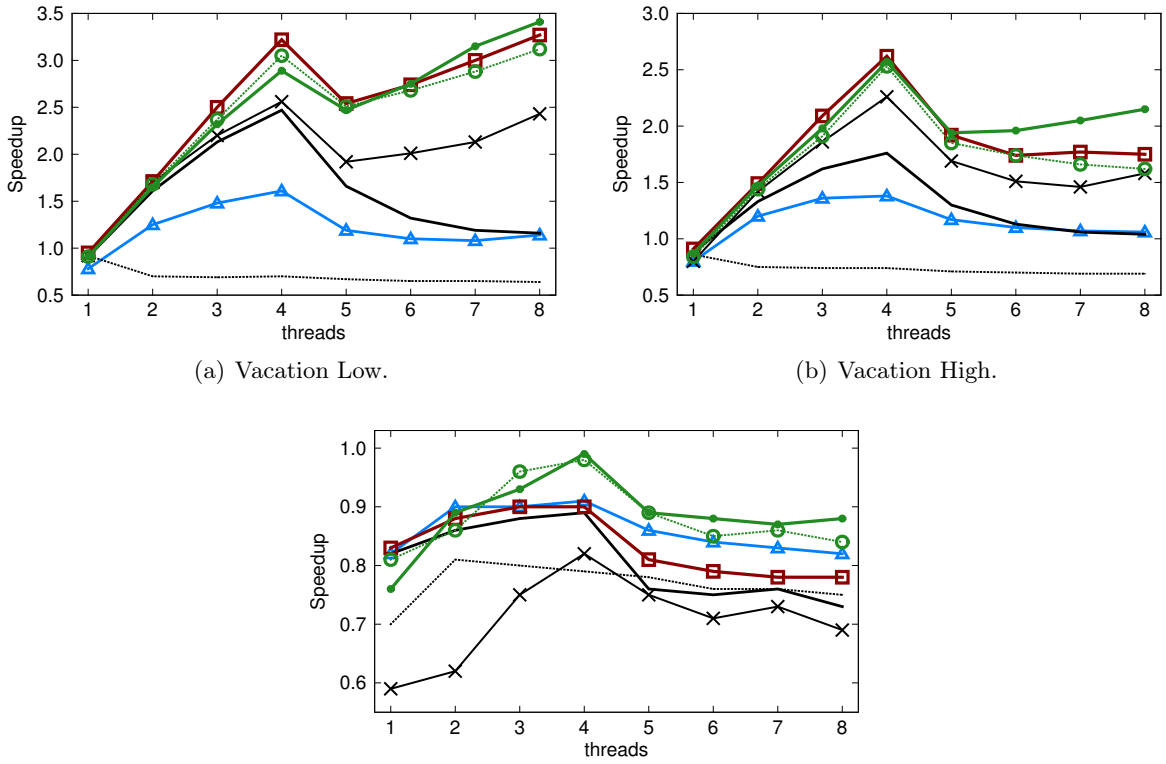
(c) SSCA2 (Kernel 1).

(d) Genome.

(e) Intruder.

(f) Labyrinth.

Figure 5.9: Speedups relative to sequential execution in the STAMP benchmarks when tuning **RTM-SGL** with different approaches (1/2).



(a) Vacation Low.

(b) Vacation High.

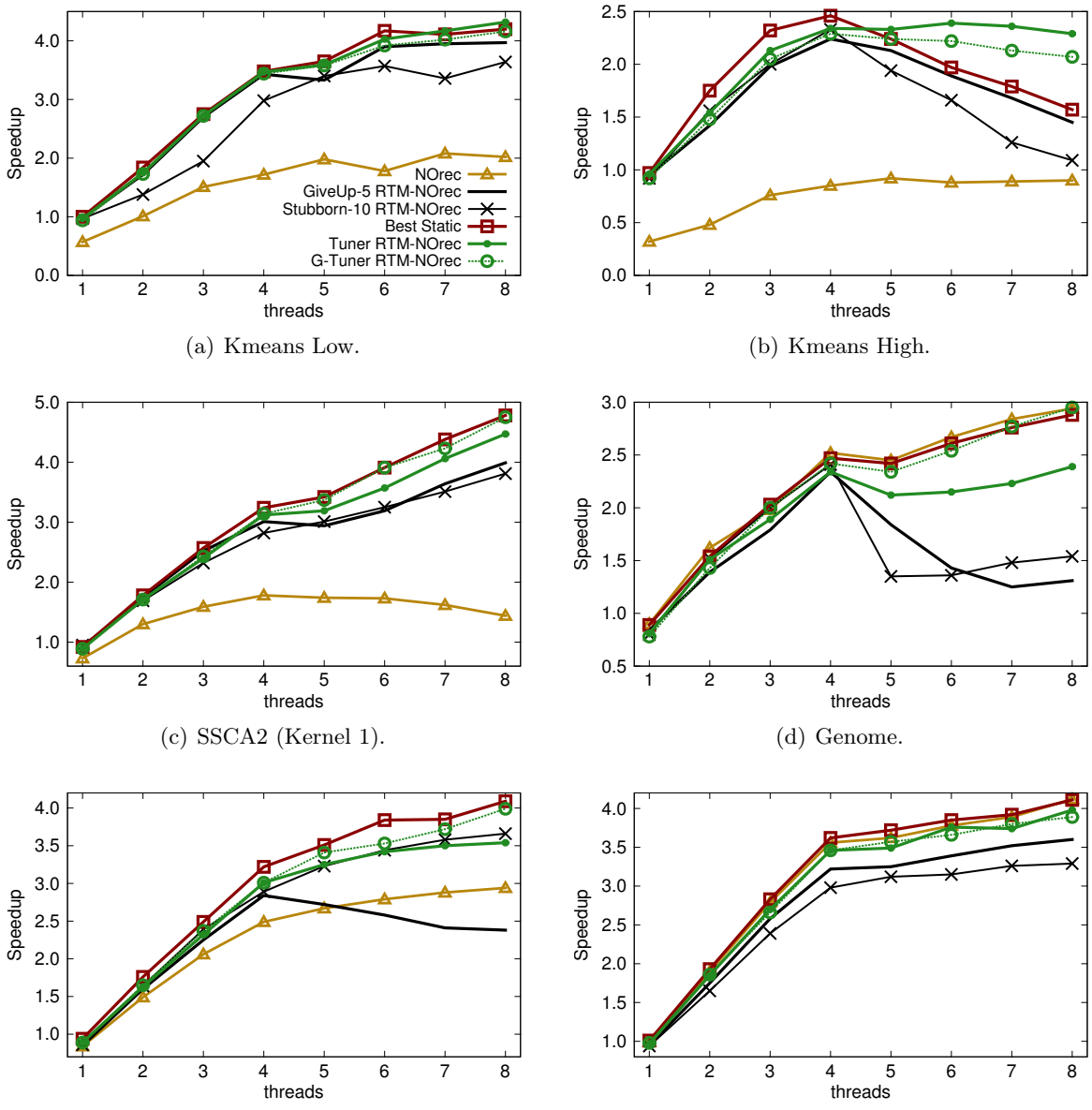
(c) Yada.

Figure 5.10: Speedups relative to sequential execution in the STAMP benchmarks when tuning **RTM-SGL** with different approaches (2/2).

Once again, we present our study with respect to both RTM-SGL (in Figures 5.9 and 5.10)[4] and RTM-NOrec (in Figures 5.11 and 5.12). In general this large set of experiments indicates a consistent gap in performance between the static configurations and the best possible variant. This gap is usually more noticeable as the concurrency degree increases — as we can see for instance in Intruder (in Figure 5.9(e)) — which is expected, since that is when the configuration parameters matter most to decide when it is profitable to insist on the hardware transactions of RTM. In short, these gaps in performance between the static alternatives and the best variant possible correspond exactly to the room of improvement that we try to explore with our self-tuning approaches in this dissertation.

In fact, both TUNER and G-TUNER are able to achieve performance improvements in all benchmarks with the exception Labyrinth, in which it yields roughly the same performance as

[4]Note that in Labyrinth (Figure 5.9(f)) the GCC and Best Static lines are overlapping, as the GCC heuristic gives up very easily, making it a nice fit for the long running transactions of this benchmark that lead to regular hardware capacity aborts.



(a) Kmeans Low.

(b) Kmeans High.

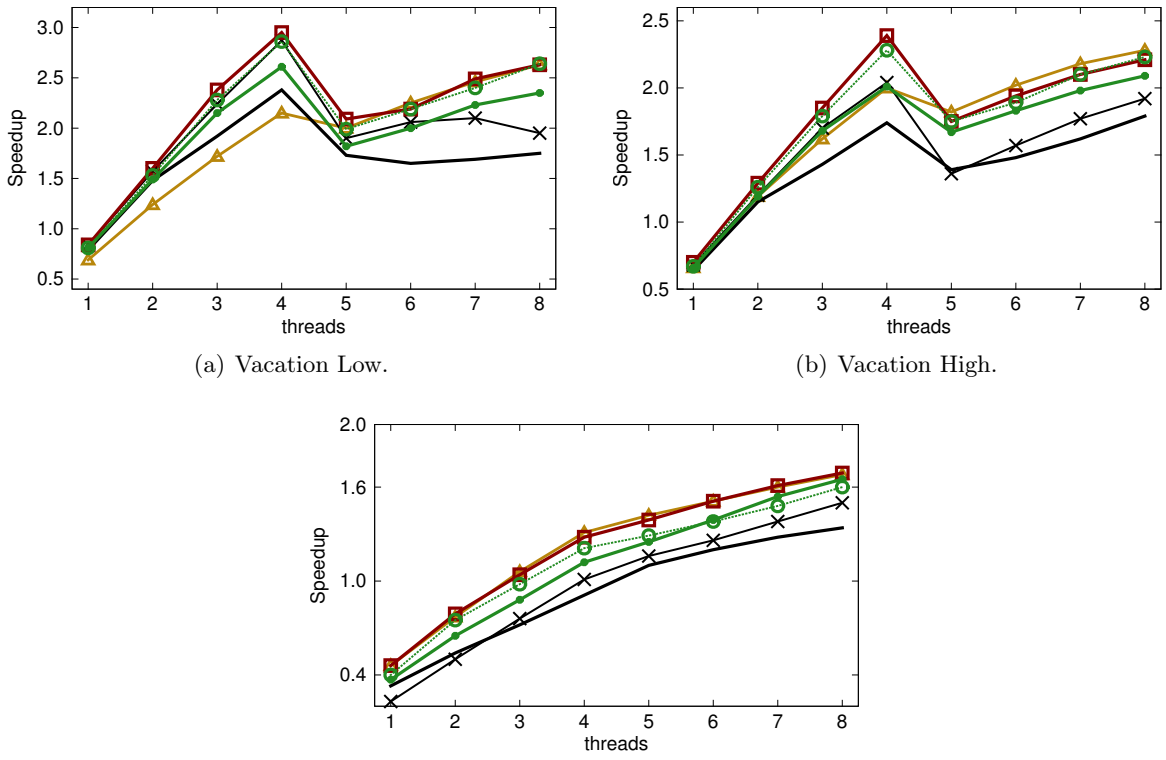
(c) SSCA2 (Kernel 1).

(d) Genome.

(e) Intruder.

(f) Labyrinth.

Figure 5.11: Speedups relative to sequential execution in the STAMP benchmarks when tuning **RTM-NOrec** with different approaches (1/2).



(a) Vacation Low.

(b) Vacation High.

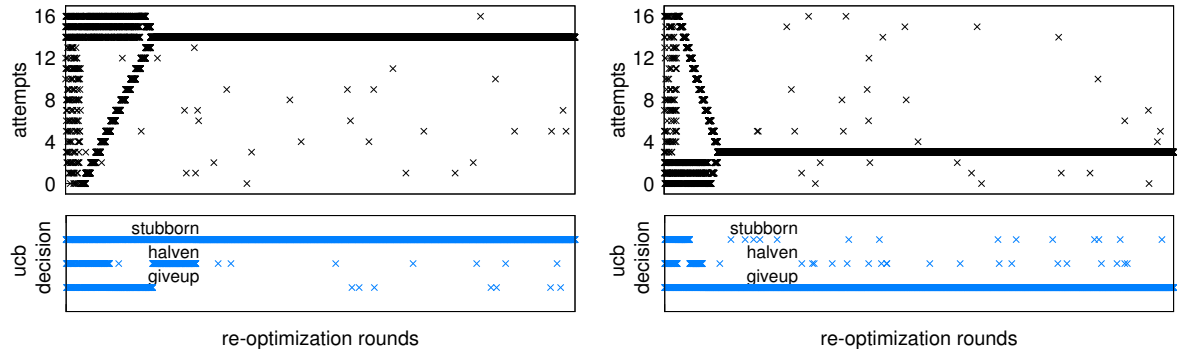
(c) Yada.

Figure 5.12: Speedups relative to sequential execution in the STAMP benchmarks when tuning **RTM-NOrec** with different approaches (2/2).

the static approaches. Note that, in Labyrinth, transactions are always too large to execute in hardware, and the benchmark executes about five hundred such large operations, which means the length of the transaction dominates the benchmark and no noticeable performance changes exist with regard to different configurations that do not insist too much on the hardware.

It is also interesting to see that there are some experiments where the self-tuning approaches perform better than what we call the Best configuration. This is a consequence of the Best configuration being devised from static configurations; we choose the one that performs best among all those that are possible. In contrast, our self-tuning approaches change the configuration during the execution, possibly obtaining a combination of configurations at runtime that performs better than any single configuration. We can observe this phenomenon in Kmeans, Vacation and Yada.

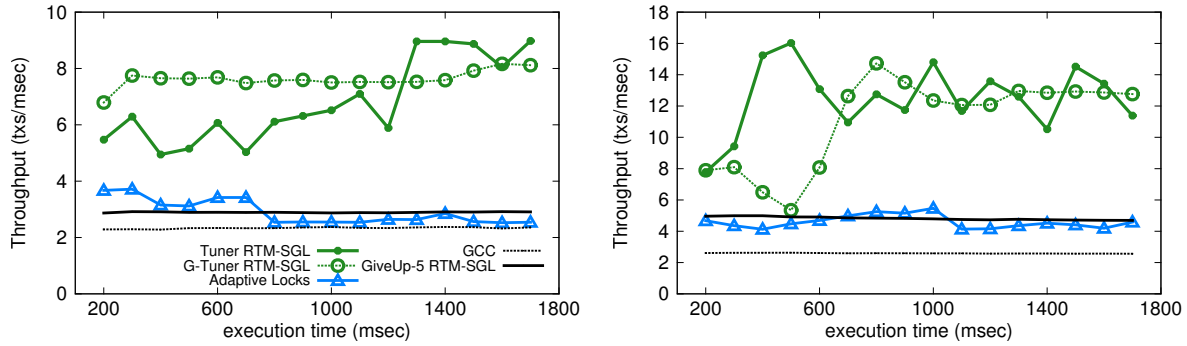
Finally, we present an example of the adaptation performed by TUNER in Figure 5.13 in the Yada benchmark (we show the adaptation of one thread among 8 running concurrently).



(a) Yada atomic block A.

(b) Yada atomic block B.

Figure 5.13: Exploration and adaptation of TUNER on two different atomic blocks.



(a) Genome.

(b) Vacation Low.

Figure 5.14: Throughput variance of each algorithm during the execution of 2 benchmarks.

There, we can see the configuration of two atomic blocks being re-optimized, and converging to two drastically different configurations: the left block executes efficiently with RTM whereas the right one does not. This illustrates the advantages that we have previously mentioned for TUNER: 1) the adaptation allows heterogeneous threads and atomic blocks to converge to different configurations; 2) an atomic block, such as that in Figure 5.13(b), can still insist moderately on using RTM as long as capacity aborts do not occur, but react quickly in case they appear. In contrast, G-TUNER uses a slightly more efficient profiling approach (with less redundant work across the threads) at the cost of having a single optimizer for all the application. As such, this results in less performance in cases such as Yada, where it is possible to benefit from the heterogeneity of the atomic blocks.

To better show this duality among our self-tuning approaches we show, in Figure 5.14, the throughput of the different algorithms over time in two benchmarks for a sample of time in their initial execution (with 8 threads). It is interesting to see that G-TUNER tends to

stabilize quicker than TUNER, although possibly at a lower throughput rate. TUNER produces a spikier throughput due to the interactions among different threads and different atomic blocks exploring the different configurations, which has an impact in the stabilization of the algorithm. Nevertheless, this has the advantage that, in some cases, it may achieve an average performance that is higher than G-TUNER (as is the case for Vacation in Figure 5.10(a)).

In contrast, we can see that the static baselines have a steady performance, albeit far worse in absolute terms. Adaptive Locks acts accordingly to an analytical model, whose impact in the configuration choice is visible as performance varies. However, its constant overheads are too high for those explorations to pay off.

5.10.4 Evaluating Energy Consumption

To conclude our evaluation study, we also assessed the energy efficiency of the different algorithms. So far we have focused on optimizing performance, but shifting the focus towards energy saving may, at least in principle, dictate a different optimization. To a large extent, the trade-off between optimizing for energy or time is a consequence of the trade-off between the usage of locks in the software fall-back and their absence in the hardware path.

However, optimizing for energy consumption is not an easy task to implement. There are commodity facilities in recent Intel processors, called RAPL [David et al., 2010], which we previously used in our comparative study in Chapter 3.

One problem is that the refresh rate of RAPL is not very high — on the order of milliseconds. Adding to this, RAPL uses model specific registers, which are exposed through the file system and require expensive calls to read and compute the energy consumed. As a result, these problems mean that measurements must be conducted over large periods of time, so that the limitations are avoided and costs are amortized, when compared to the exploration/exploitation optimizations that we propose in this dissertation.

In fact, we had seen that performance and energy followed similar trends, during the comparison in Chapter 3. We now seek to assess that correlation in more detail, and considering also the full space of possible configurations for the fall-back of RTM (whereas before we had considered only a static configuration for our comparative study).

Given these challenges, which seem hard to overcome with current commodity hardware, we

Table 5.8: Distance correlation between performance and energy consumption averaged over the runs with different number of threads for each benchmark. Values closer to 1 show dependence between performance and energy consumption.

Benchmark	Correlation	Benchmark	Correlation
genome	0.74	linked-list low	0.91
intruder	0.84	linked-list high	0.87
labyrinth	0.82	skip-list low	0.94
kmeans high	0.76	skip-list high	0.81
kmeans low	0.92	hash-map low	0.98
ssca2	0.97	hash-map high	0.72
vacation high	0.55	rbt-low	0.95
vacation low	0.74	rbt-high	0.73
yada	0.77	*average*	0.81

hypothesized the following: can we optimize a TM application in terms of performance and, as a side effect, also optimize the application in terms of energy efficiency?

To answer this question, we conducted a series of experiments. As a first experiment, we took every execution in all our benchmarks and measured the distance correlation between the time to complete the benchmark and the energy spent in doing so. These executions encompass all the possible configurations, benchmarks and parallelism degree, which amounts to almost twenty five thousand runs. We used a state of the art distance correlation metric [Székely et al., 2007] in which two random variables are considered dependent if the distance is 1, and independent if the distance is 0.

In Table 5.8 we show the computed distance correlation between time and energy. The objective is to assess the extent to which these two variables are related. The results are shown for each benchmark, and averaged across all the configurations and degrees of parallelism. As a result, we obtain an average correlation of 0.81, with an outlier in Vacation High with 0.55 and all others above 0.70. This suggests a relatively strong correlation between the energy and performance achievable by any configuration considered in the study.

The data in Table 5.9 provides an alternative, interesting perspective from which to analyze the correlation between energy and performance. The experiment whose results are reported in this table was designed to answer the following question: how distant is energy consumption from the optimum in the configuration selected by (G)-Tuner, which, we recall, uses as target metric for the self-tuning process a performance-related metric?

Table 5.9: Relative energy of the best configurations, aimed for performance, with respect to the best configuration in terms of energy. Values closer to 1 show that optimizing for performance also optimizes for energy consumption. We show the geometric mean across different number of threads for each benchmark.

Benchmark	Relative Energy	Benchmark	Relative Energy
genome	0.99	linked-list low	1.00
intruder	1.00	linked-list high	1.00
labyrinth	0.92	skip-list low	1.00
kmeans high	1.00	skip-list high	0.98
kmeans low	1.00	hash-map low	0.99
ssca2	1.00	hash-map high	0.99
vacation high	0.99	rbt-low	1.00
vacation low	1.00	rbt-high	1.00
yada	0.89	*average*	0.98

For each benchmark and parallelism degree, we took the best performing configuration in terms of time (configuration T) and energy (configuration E), which are typically different (i.e., we verified that normally $T \neq E$). Then, we compare the relative energy obtained with T with respect to that of E (i.e., the optimal one). As such, we obtain the relative loss in terms of energy when optimizing for time compared to that if we optimized for energy.

The results, shown in Table 5.9, show with an outstanding consistency that the loss is negligible. The average relative energy consumption (i.e., a metric akin to that of speedup that we used earlier) shows a value of 0.98, which means that for the most part we obtain the optimal energy when focusing on performance alone.

This allows us to conclude also that our self-tuning proposals benefit both metrics of time and energy together, while only focusing on the former.

5.11 Summary

In this chapter we shed light on one issue that can have a great impact on the performance of the recent Intel RTM: the interplay with the software fall-back that regulates how to cope with failed hardware transactions. We showed that the optimal tuning of the software policy that regulates the re-execution of hardware transaction is strongly workload dependent, and that the relative difference in performance among the various possible configurations can be remarkable

(up to 10×).

Motivated by these findings, we presented a novel self-tuning approach that combines reinforcement learning techniques and hill climbing exploration-based algorithms to self-tune RTM in a workload-oblivious manner. This means that our proposal does not require a priori knowledge of the application, and executes fully online, based on the feedback on system's performance gathered by means of lightweight profiling techniques.

We proposed two designs, called TUNER and G-TUNER, which we integrated with the well known GCC compiler, thus achieving total transparency for the programmer. The two designs obtain similar average results, although our study highlighted interesting trade-offs that make TUNER best fit for RTM with a single-global lock fall-back, and G-TUNER for RTM with an STM as the fall-back.

Our extensive evaluation study also showed consistent average gains of 60% over the best static alternative suggested by previous studies to regulate the RTM's retry policy. Furthermore, our proposed solutions had performance averaging 5% less from the optimal performance among all possible configurations of RTM within the ranges that we considered. Finally, we concluded also that our proposals benefit equally energy-consumption, as energy efficiency and performance appear to be strongly correlated in the large majority of the workloads encompassed by our benchmarks' test-bed.

6 Seer: A Probabilistic Scheduler for HTM

In the previous chapter we have addressed the correct tuning of the retry policy for best-effort HTMs, using as driving case study the Intel RTM. However, the retry policy can only provide benefits so long as the hardware transactions abort up to a limit, as otherwise they are prone to eventually triggering the fall-back path.

In this chapter we take a different look at the challenges underlying concurrency control with the recent best-effort HTMs. The key idea here is to attempt to reduce the aborts of hardware transactions, i.e., an objective similar to that of TIME-WARP in Chapter 4, yet using a different technique: restricting the parallelism of transactions that have a high risk of aborting due to concurrent interactions with each other, by scheduling them in mutual exclusion.

As such, this complements the previous proposal of TUNER, as aborts are reduced in a best-effort fashion, thus still causing HTM to have widely varying optimal retry policies.

The technique presented in this chapter augments the software that manages the HTM invocations (whose internal logic remains the same; i.e., that of commodity processors such as Intel's Haswell), similarly to TUNER, which means that the idea is not to change or propose a new HTM *per se*. This contrasts with TIME-WARP, which reduced aborts by using a new STM algorithm, whose concurrency control was changed with respect to other more traditional approaches.

6.1 The Problem

Due to the speculative nature of TM implementations, transactions are likely to be restarted and aborted multiple times in conflict prone workloads. This has motivated a large body of research on **schedulers** (e.g., [Yoo and Lee, 2008, Dragojević et al., 2009b, Dolev et al., 2008]), whose key idea is to serialize the execution of transactions that are known to generate frequent aborts. To enable this, the application is typically instrumented to provide identifiers associated

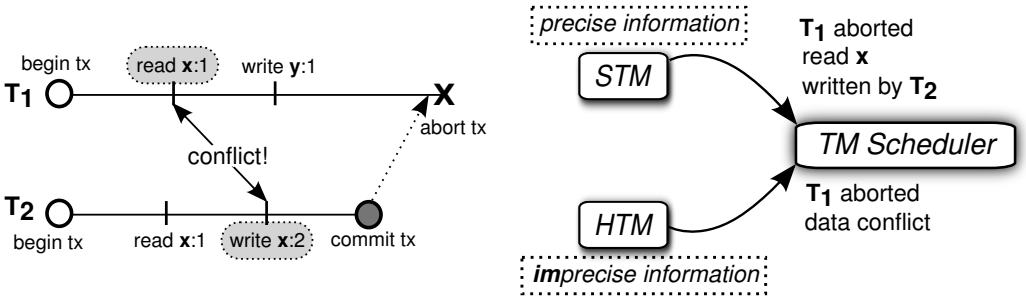


Figure 6.1: Two transactions causing a conflict. The information returned by the TM varies depending on its nature: STMs are able to precisely identify the source of the abort, whereas commodity HTMs provide only a coarse categorization of the abort.

with the call to start and end a transaction, thus mapping them to atomic blocks in the source code. Then the scheduler acts at the level of such atomic blocks, by allowing (or not) instances of transactions of different types (i.e., atomic blocks) to proceed in parallel.

However, most existing scheduling techniques were designed to operate with STMs: they rely on specific support provided by the STM to gather knowledge on the conflicts that occurred between transactions. Typically, upon a transaction abort, the STM library can report back to the scheduler which specific memory access and concurrent transaction dictated the abort [Rito and Cachopo, 2014, Dragojević et al., 2009b]. This happens because that software library has full knowledge of the read and write-sets of the transactions, and can be easily extended to externalize such information back to the scheduler. This is illustrated in Figure 6.1, where we depict transaction T_1 aborting due to a read-write conflict with a concurrent transaction T_2. An STM library is able to report this precise information back to a TM scheduler.

With the adoption of HTMs such as that of Intel RTM, however, we lose much of this ability. When a hardware transaction is aborted, the feedback is limited and insufficient to pinpoint which transaction caused the abort. As shown already in Chapter 3, and exemplified in Figure 6.1, these HTMs merely distinguish between a data conflict and other abort causes (e.g., exceeding hardware buffers). As seen also in Section 2.1, this means that calls to begin a transaction either report that the transaction is started, or that some error occurred and caused it to abort (namely, capacity overflow of the processor caches, conflict with another undisclosed transaction, or some other problem due to the limitations of HTM).

For this reason, schedulers for STMs, which assume exact knowledge on the conflict patterns

among transactions, cannot be straightforwardly employed with HTM, which are instead only capable of providing *imprecise information*. The problem is then how to devise a scheduler that can work with that very limited information and still make accurate decisions. This challenge is exacerbated by the strong impact that schedulers can have on performance: while appropriate scheduling decisions can provide better performance, the contrary is also true, in that erroneous scheduling (e.g., caused by inaccurate/incomplete knowledge) can degrade performance severely by unnecessarily restricting the available parallelism.

6.2 Overview

In this chapter we propose SEER, the first scheduler (to the best of our knowledge) to address the HTM limitations discussed above. The key idea of our proposal is to gather statistics at runtime to detect, in a lightweight but possibly imprecise way, the set of concurrently active transactions upon abort and commit events.

This information is used as input for an on-line inference technique that uses probabilistic arguments to identify conflict patterns between different atomic blocks of the application in a reliable way, despite the imprecise nature of the input statistics. The final step consists in exploiting probabilistic knowledge on the existence of conflict relations to synthesize a fine-grained, dynamic (i.e., possibly varying over time) locking scheme that serializes "sufficiently" conflict-prone transactions.

A noteworthy feature of SEER is that it relies on reinforcement learning techniques to self-tune the parameters of the probabilistic inference model. To this end, SEER relies on a probabilistic hill climbing technique that explores the configuration space of the model's parameter, while gathering feedback at runtime about the application's performance and accordingly adjusting the granularity of the locking scheme. Indeed, an appealing characteristic of this dynamically inferred locking scheme is that it does not need to be perfect (e.g., it can suffer of false negatives) in capturing conflicts between atomic blocks of the application, since correctness for transactions is still enforced by the underlying HTM.

SEER includes also an additional novel mechanism that is designed to address another performance pathology of existing HTM systems: when multiple hardware threads are concurrently active on the same physical core, the likelihood of incurring aborts due to capacity exceptions

can grow to such an extent that it can eventually cripple performance. This is a direct consequence of the fact that the information used by the HTM concurrency control algorithm is entirely stored in the CPU caches, which may be shared by hardware threads running on the same core. SEER copes with this issue by introducing a simple, yet effective abstraction, the *core lock*, which serializes the execution of hardware threads that share the same core when capacity exceptions are detected.

Besides reducing aborts due to conflicts over the accessed memory regions, SEER achieves also a drastic reduction of the frequency of activation of the software fall-back path of the HTM system, whose sequential nature is known to hamper HTM performance [Yoo et al., 2013, Jacobi et al., 2012].

Overall, our experimental study shows that, by applying SEER to standard TM benchmarks, one can obtain gains up to 3.6× and average speedups of 65% across various degrees of parallelism in a processor with 28 cores and HTM support. We also confirmed that our design yields more accurate probabilities than if we used other alternative calculations that use the same input data.

The rest of this chapter is organized as follows. In the following section we discuss the state of the art in TM schedulers and identify the key factors that make our proposal novel. Then, in Sections 6.4-6.5, we present the details of our SEER implementation. We then validate its design choices for the probabilistic inference scheme in Section 6.6. Finally, we evaluate our proposal in Section 6.7 by comparing it with several alternatives, and conclude in Section 6.8.

6.3 Related Work

There is a rich body of work on TM schedulers. Next, we survey the state of the art in this area, and then summarize the characteristics of existing systems, contrasting them with our novel proposal SEER.

Roughly speaking, the objective of a TM scheduler is to decide when it is best to execute a transaction, possibly deciding to serialize concurrent transactions based on their likelihood of contending with each other, with the ultimate goal of maximizing performance (typically throughput).

Most of the existing schedulers target STM systems, which are assumed to be able to provide

precise information on the conflicts that caused the abort of a transaction. This is the case for CAR-STM [Dolev et al., 2008] and Steal-On-Abort [Ansari et al., 2009], where there are N serialization queues (one for each thread), and an aborting transaction T_i is placed in the queue of T_j that caused its abort. The idea is that T_i is serialized after T_j because it shall be executed by the thread currently running T_j, with which it conflicted. Both these schedulers were proposed in the scope of STMs, which were extended to obtain precise information on aborts.

Steal-On-Abort, although initially implemented in software, was later also proposed for an HTM simulator [Ansari et al., 2010] by extending LogTM-SE [Yen et al., 2007] and running on a SPARC simulated machine. However, this work assumed hardware extensions to support enqueuing the serialized transactions in each core of the processor. The current expectation is that manufacturers, such as Intel and IBM, will be quite resistant to changes in the hardware due to its complexity [Jacobi et al., 2012] and cost of verification [Adir et al., 2014]. Hence, it is particularly relevant to devise a scheduling solution for *current* HTMs: one that operates in absence of accurate information on the conflict patterns among transactions, like SEER does.

More recently, ProPS [Rito and Cachopo, 2014] followed a similar approach to the ones above but, instead, focused on long running transactions: each abort event is used to accumulate a contention probability between every pair of transaction types (i.e., atomic blocks); whenever a transaction T is about to start, it may have to wait in case there is an atomic block being executed in a concurrent transaction that is expected to conflict with T with high probability. This approach also requires precise information to guide the scheduling decision, which is not the case for HTMs such as Intel's HTM. Shrink [Dragojević et al., 2009b] was originally published before ProPS, but its idea is the same, with the addition that it is fed with past history of transactions' read- and write-sets: assuming there is some data accesses locality between transactions' restarts, the scheduler uses this information to predict conflicts that would happen if the transaction were allowed to run against current concurrent transactions. Such fine-grained information is not available in HTMs, and could only be made available via additional software instrumentation, yielding considerable overheads.

TxLinux [Rossbach et al., 2007] and SER [Maldonado et al., 2010] both changed the Linux scheduler to be transaction-aware, the difference being that the former was integrated in a simulated HTM called MetaTM and the latter was fully in software. Similarly to the other works, these proposals also require precise information.

Contrarily to the schedulers above, ATS [Yoo and Lee, 2008] works with imprecise information, i.e., coping with the lack of knowledge on which pairs of transactions conflict during their execution. ATS maintains a contention factor in each thread, updated when transactions abort and commit, such that a single lock is acquired when contention exceeds a specified threshold. This simple approach is agnostic of the atomic blocks being executed, as the whole problem is subsumed by a single contention factor. The positive side is that it works with currently available HTMs. In fact, this is the *de facto* technique used with commodity HTMs due to their best-effort nature, discussed in earlier chapters, and for which we recall: because no transaction is guaranteed to commit, a software fall-back must be provided to ensure progress; the single lock fall-back that is typically used [Yoo et al., 2013, Jacobi et al., 2012, Matveev and Shavit, 2013] is, in essence, akin to ATS. Since ATS relies on a single contention factor and one lock for serialization, it alternates between serializing all transactions or letting them all execute concurrently; hence, that is why we characterize it as a a coarse-grained scheduler.

To overcome the very simplistic approach of ATS, there is the recent proposal of the Octonauts scheduler [Mohamedin et al., 2015b], in which there are fine-grained locks (in the form of serialization queues) for different objects/variables in the program (or, conceptually, different memory regions). To achieve this, Octonauts requires a priori static information about the application/workload: to schedule a transaction, it uses static knowledge about the working-set of a transaction (i.e., its read- and write-sets), so that it can place the transaction in the queues for the objects to which it will read and write. This effectively serializes the transaction with concurrent conflicting transactions. To cope with incorrect static information, Octonauts still uses the HTM support to enforce correctness — similarly to our approach with SEER, in which our devised locking scheme need not be safe in terms of the concurrency control. Evidently, this approach suffers from the major drawback in that the programmers must provide this static information about the transactions' working set. This represents a non-negligible source of complexity for programmers, hence contradicting one of the main motivations of TM, i.e., to simplify concurrent programming. Conversely, SEER extracts any necessary information regarding conflict patterns among transactions in a runtime and completely transparent fashion to the programmers.

We summarize the above state of the art in Table 6.1. Briefly, we can see that most schedulers cannot cope with imprecise information, as they rely on knowledge about the pair of transactions that are involved in a data conflict, possibly along with information about the memory address

Table 6.1: Comparison of TM schedulers in terms of: regulating an STM and/or HTM, working without precise information on which transaction caused the abort, whether it uses multiple fine-grained locks to schedule transactions' execution, and whether it does not require static information from the workload. SEER, our proposal, is the only scheduler that provides all the following properties: 1) works with HTM; 2) does not require precise feedback on aborts; 3) adopts a fine-grained serialization mechanism; and 4) does not require static information about the workload.

Scheduler	SW	HW	Imprecise Information	Fine Grained	No Static Information
ATS [Yoo and Lee, 2008]	✓	✓	✓	χ	✓
CAR-STM [Dolev et al., 2008]	✓	χ	χ	✓	✓
Shrink [Dragojević et al., 2009b]	✓	χ	χ	✓	✓
ProPS [Rito and Cachopo, 2014]	✓	χ	χ	✓	✓
SER [Maldonado et al., 2010]	✓	χ	χ	✓	✓
TxLinux [Rossbach et al., 2007]	χ	✓	χ	✓	✓
SOA [Ansari et al., 2009, Ansari et al., 2010]	✓	✓	χ	✓	✓
Octonauts [Mohamedin et al., 2015b]	✓	✓	✓	✓	χ
SEER	χ	✔	✔	✔	✔

that generate a conflict. To the best of our knowledge, ATS and Octonauts are the only two schedulers that can cope with that limitation inherent to commodity HTMs. On one hand ATS provides only a coarse serialization via a global lock. On the other hand, Octonauts assumes static information about transactions' working set. As such, our contribution SEER is unique by being applicable to commodity HTMs (i.e., it works with imprecise input) and allowing to serialize multiple transactions concurrently in a fine-grained manner without assuming any a priori knowledge on the application's workload.

6.4 Scheduling HTM Transactions with SEER

Schedulers for TM systems, independently of their software or hardware nature, benefit particularly from the availability of fine-grained precise information about what causes the abort of a transaction. This means that if we are executing a transaction for an atomic block of our program, and we know that it aborted due to a concurrent transaction executing another specific atomic block, then it is probably better to avoid executing transactions for those atomic blocks concurrently.

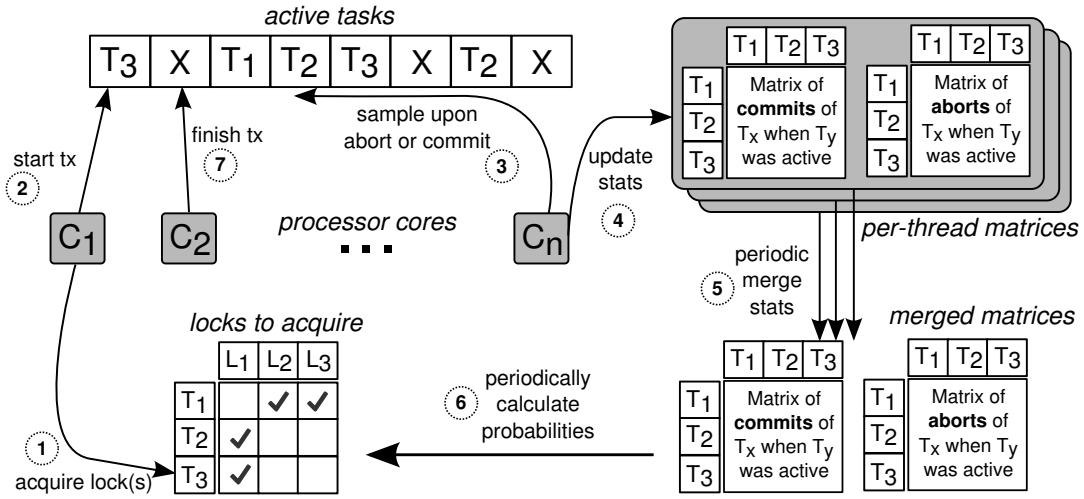


Figure 6.2: Architecture of the components used in SEER. The idea is to assess the probability of conflicts between transactions, without requiring precise information by the HTM. To do so, transactions are announced right before they are executed on a given core, and then this information is sampled upon commits and aborts, to compensate for the lack of feedback from the HTM. While not totally accurate, this information allows to probabilistically infer the relevant conflict patterns among transactions over time, and then to produce a dynamic locking scheme that serves to schedule transactions (by preventing some transactions from running concurrently).

Having access to such information is typically trivial in STMs. However, as discussed earlier in this chapter, mainstream HTMs provide little to no feedback with respect to this matter. In particular for Intel's HTM, upon the abort of a hardware transaction, it is possible to know only a rough categorization: for instance, whether it was a data conflict; or whether the space available for the read- or write-set buffers in the hardware caches was exhausted; or whether there was an interrupt that caused a context switch or a ring transition. As such, no information is given about which transaction was the cause for the abort. This is the challenge that prevents existing schedulers from being effectively applicable to existing HTMs.

The high level idea of our solution is to take a probabilistic approach. While we do not know what exactly causes a transaction T_i to abort, because the HTM provides no such information, we can try to infer the answer by observing enough times which transactions were active when T_i aborted. By repeating this observation over time, we can gather probabilistic knowledge on the likelihood of conflicts between pairs of transactions. This knowledge can then be exploited to decide, when a transaction starts, whether to schedule it or not depending on the conflict probabilities with the currently active transactions.

The probabilistic inference mechanism of SEER is based on three key ideas: (1) we continuously collect runtime information about the transactions concurrently active upon commit and abort events, by means of a lightweight, synchronization-free monitoring mechanism; (2) we periodically analyze this information and estimate probabilities of aborting/committing in the presence of other specific transactions; and (3) this information is used to periodically devise a fine-grained locking-scheme, whose locks are acquired upon the start of a transaction and allow for serializing the execution of conflict prone-pairs of transactions (without blocking other transactions not likely to incur any conflict).

Figure 6.2 portraits the life-cycle of transactions within our scheduler. The objective of this life-cycle is to populate a global table that reifies the automatically inferred locking scheme. SEER uses one lock for each transaction in the target application (identified in the columns). Each row i of the table specifies the locks that transaction T_i should acquire, indicating that T_i conflicts often with the transactions associated with these locks, and that these transactions should not be executed concurrently. We associate each atomic block in the application source code to a different transaction T_i: this way, we seek to serialize transactions with a fine granularity, contrarily to other approaches that work with HTM and that use a single lock for serialization [Yoo and Lee, 2008].

To understand how to reach that objective, we begin by describing the life-cycle of SEER. In step ①, a transaction T_3 is about to be executed on core C_1. Before doing so, it acquires the locks defined by SEER in a global table: in this case, lock L_1. Then, it announces that C_1 is executing T_3 in the list of active transactions in step ②. Step ⑦ shows that transactions are removed from that list when they are finished.

By acquiring lock L_1 in the lock table, this means that transaction T_3 was deemed to contend with T_1. Although instances of T_1 do not acquire lock L_1, because they do not contend with 'themselves', they do co-operate with contending transactions (such as T_2 and T_3) by waiting for their completion, before starting executing, if lock L_1 is found to be taken. In general:

1. A transaction T_x **waits** for its lock L_x to be free before proceeding, which serves to respect our scheduling policy.

2. It **locks** L_y if it contends with T_y (we allow $x = y$, in which case T_x contends with instances of itself).

We are left with describing how the locking scheme is generated. Step ③ illustrates that, upon a commit or abort of a transaction T_n running on core C_n, the active transactions list is sampled [1] and the transactions found there are incremented in two per-thread matrices, namely commitStats and abortStats, which are stored as thread-local variables (step ④). An entry x, y in commitStats (resp. abortStats) tracks the frequency of commit (resp. abort) events for transaction T_x, in which T_y was found to be running (in the active transactions list), after the commit (resp. abort) of T_x.

Consider for instance that T_1 only conflicts often with T_3. Such fact is unknown beforehand and our approach aims to infer it in runtime: as we gather statistics, over time, recurrent events emerge and become identifiable using probabilistic inference.

Periodically, these statistics are merged, across all per-thread's matrices, into two global matrices in step ⑤. These are used to calculate and update the locking scheme to reduce aborts of transactions. The intuition is to use the information about how often T_x committed and aborted in the presence of each different transaction. The challenge in doing so is to identify, among all captured conflicts, which ones occur frequently enough to benefit from throttling down concurrency. The ability to extract these decisions using solely the imprecise information provided by commodity HTMs is what makes Seer novel with respect to other schedulers.

As a result, we are able to periodically generate a dynamic locking scheme, as depicted in step ⑥. As explained above, these locks are used to serialize transactions with a fine granularity. This is a key feature that allows Seer to yield substantial performance improvements as we later show in Section 6.7. Another noteworthy feature of Seer is that it works in a completely transparent fashion to the programmer. We require only minimalist compiler support, by enumerating the atomic blocks in the program, and passing their unique identifier (one per source code atomic block) into the TM library calls. The scheduler itself is implemented in the TM library that regulates the software fall-back management. Further, Seer fully automates the tuning of internal parameters in the probabilistic inference, via a self-optimization mechanism that is driven by the feedback gathered at runtime on the throughput of the TM system.

Finally, Seer introduces the abstraction of *core locks*, i.e., locks that prevent the concurrent execution of multiple hardware threads on the same physical core. The idea of core locks is based

[1] As we shall see, this sampling may range from scanning a full snapshot of the list, to obtaining only one uniformly random concurrent transaction.

Table 6.2: Characterization of the data-structures used in SEER. Some of these are visible in the architecture shown in Figure 6.2, whereas the rest is used in Algorithms 15-19.

Variable	Description
thread	Per-thread structure to hold metadata during the execution of a transaction.
sgl	Single-global lock used in the software fall-back path of the HTM.
activeTxs	Global array where threads announce the transactions they are executing.
commitStats	Global matrix where, each line for transaction T_i, reports the transactions that were concurrently running whenever T_i committed. This matrix is periodically built by summing the per-thread equivalent matrices kept in each thread variable.
abortStats	Similar to commitStats, but for abort statistics.
executions	Array with total number of executions (commits and aborts) of each transaction.
locksToAcquire	Global matrix where each line corresponds to a transaction and the columns define the locks that should be acquired for the transaction according to SEER.
txLocks	Global array of locks, one per transaction (i.e., atomic block) of the program.
coreLocks	Global array of locks, one per core of the processor.

on the observation that, in workloads characterized by frequent transactions with non-minimal memory footprints, the likelihood of capacity aborts in the HTM is exacerbated when multiple threads are allowed to execute freely, as they contend for the shared caches of the core.

6.5 Detailed Algorithm

We now present the detailed description of SEER. We first list the data-structures and metadata used by SEER, in Table 6.2, most of which were already referred in abstract terms when presenting the architecture in the previous section.

Conventional HTM usage. In the algorithms explained next we shall use as starting point the conventional HTM usage that we described initially in Section 2.1. We highlight lines, which are associated with the conventional HTM mechanisms, with a $\triangle$ (other lines belong to SEER). Also, we note that now the HTM_START and HTM_END procedures' interface was augmented to receive the information that a given transaction $txId$ is initiated by a given *thread*.

Also, we recall that the HTM_START procedure implements a retry loop to try to execute a hardware transaction, up to some threshold (`MAX_ATTEMPTS`), resorting to a fall-back path in

Algorithm 15: SEER algorithm.

```
 1: HTM_START(thread, txId)
 2:   thread.core ← current-core()                          ▷ thread is bound to core
 3:   thread.acquiredTxLocks ← false
 4:   thread.acquiredCoreLock ← false
 5:   activeTxs[thread.core] ← txId
6△    attempts ← MAX_ATTEMPTS
7△  begin:                                                  ▷ used to jump to and re-attempt with HTM
 8:   WAIT-SEER-LOCKS(thread, txId)
9△    htmStatus ← _xbegin()
10△   if htmStatus = _XBEGIN_STARTED
11△     if is-locked(sgl)                                   ▷ ensure correctness with fall-back
12△       _xabort()
13△     else
14△       return                                            ▷ hw transaction enabled, proceed to tx
15△   ▷ hw transaction aborted, handle before restarting
16:   REGISTER-ABORT(thread, txId)
17△   attempts ← attempts - 1
18△   if attempts = 0                                       ▷ give up on HTM, fall-back to lock
19:     RELEASE-SEER-LOCKS(thread, txId)
20△     acquire-lock(sgl)                                   ▷ SW fall-back with a single lock
21△     return                                              ▷ SW fall-back path taken, proceed to tx
22△   ▷ before re-attempting, trigger our scheduler SEER
23:   ACQUIRE-SEER-LOCKS(thread, txId, htmStatus)
24△   goto begin
```

case the threshold is reached (in line 20). Note that the function to begin a hardware transaction, *_xbegin()* (in line 9), returns a status that normally represents that the transaction has started, i.e., the predicate in line 10 evaluates to true. Otherwise, this status indicates a coarse categorization of the abort. An aborted hardware transaction transparently jumps back, and returns from this function, akin to the *setjmp*/*longjmp* mechanism used in C/C++.

Seer Algorithm. We now discuss the various mechanisms that augment this conventional HTM usage with SEER, briefly: i) transactions are announced to other cores (see line 5), ii) aborts are registered in the per-thread statistics (see line 16), and iii) locks are used to induce fine-grained serialization between contending transactions (see lines 8 and 23). We present each part next.

The HTM_END procedure is presented in Algorithm 16 where we finish the hardware transaction, or release the global lock, depending on the path taken in START. In case the transaction was successfully committed via a hardware transaction, we add this information to our per-thread statistics in line 28, and possibly release locks acquired by our scheduler in line 29. Finally, we remove the transaction from the *activeTxs* list.

The procedures for registering aborts and commits are shown in Algorithm 17. The idea

Algorithm 16: SEER algorithm.

```
25: HTM_END(thread, txId)
26△   if _xtest()                          ▷ returns true if inside a HW transaction
27△     _xend()                            ▷ tries to commit the HW transaction
28:     REGISTER-COMMIT(thread, txId)
29:     RELEASE-SEER-LOCKS(thread, txId)
30△   else
31△     release-lock(sgl)                  ▷ executed with lock-based fall-back
32:   activeTxs[thread.core] ← ⊥
```

is to sample the *activeTxs* list and to increase the frequency of a transaction found there, in the row corresponding to the transaction that has aborted/committed (identified by txId). This is the mechanism that we use to infer information about conflicts, and to compensate for the lack of feedback from the HTM about the pairs of conflicting transactions. In general, this collection of statistics may not be completely accurate, and could suffer of both false positives and false negatives. SEER copes with this uncertainty using probabilistic inference techniques, whose details we shall discuss shortly.

Furthermore, notice that we are sampling the active transaction of 1 thread (following a round-robin scheme), instead of scanning the full list. This is to ensure that the overheads associated with this collection of statistics remain constant and independent of the number of threads used. On top of this, we highlight also that we follow a round-robin scheme, i.e., each thread chooses a different thread every time it performs this sampling. This increases the statistical quality of the sampling in presence of threads that have to retry the same transaction often due to hardware aborts. As we shall see in our evaluation, specifically in Section 6.7.4, these two choices are of paramount importance particularly given the large number of hardware parallelism available in the machine we used.

Notice that the aforementioned statistics are maintained per-thread, i.e., in a private fashion. Furthermore, the *activeTxs* list ends up being a set of single-writer multi-reader registers; we do not place any synchronization when accessing the list, with the intent of keeping it lightweight.

The procedures for lock management, according to our scheduler, are defined in Algorithm 18. We use two types of locks:

1. **txLocks:** one per transaction of the application, to serialize contending transactions according to the probabilities (line 47) that we describe later (in Algorithm 19). Our scheduler may dictate that a transaction acquires some of these locks only when the transaction has

Algorithm 17: SEER algorithm.

```
33: REGISTER-ABORT(thread, txId)
34:   threadToSample ← getRoundRobinThread()        ▷ following a round-robin scheme
35:   if activeTxs[threadToSample] ≠ ⊥
36:     thread.abortStats[txId][activeTxs[threadToSample]]++

37: REGISTER-COMMIT(thread, txId)
38:   threadToSample ← getRandomThread()
39:   if activeTxs[threadToSample] ≠ ⊥
40:     thread.commitStats[txId][activeTxs[threadToSample]]++
```

spent most of its attempts in hardware transactions — it has one left — as a last resort measure to obtain progress before triggering the global lock in the fall-back.

2. **coreLocks:** one per physical core of the processor, to reduce capacity aborts, which are amplified due to hardware threads that share the private caches of a physical core. These caches are small and limit the size of hardware transactions, more so if shared among several. Hence, we acquire the coreLock when a capacity abort is detected (line 44).

Furthermore, we also introduce a contention avoidance technique, which imposes waiting before starting a transaction (in line 8). This is presented in WAIT-SEER-LOCKS, in Algorithm 18, where there are two main ideas. First, we use a known technique to avoid the lemming effect [Dice et al., 2008]. The problem is that hardware transactions quickly exhaust their budget of attempts when the fall-back lock is taken and tend to execute mostly in the fall-back as a consequence. To reduce this chance, a transaction waits if the global lock is taken, as otherwise it would likely abort in line 12.

The second idea behind WAIT-SEER-LOCKS is to also wait in case the txLock and/or coreLock are taken by another thread (lines 57 and 58). The intuition is that, even though this thread may not have had aborts that lead it to acquire locks, it is beneficial if it co-operates with concurrent threads that have taken the locks, giving them a chance to complete without conflicting. Doing so is instrumental for the meaningfulness of the locking scheme that we present next while avoiding a transaction to having to pessimistically always acquire the lock of its transaction.

The locking scheme is updated in line 52. At that point the thread was waiting for the single-global lock to be released, for which reason executing the logic of SEER instead is not delaying the progress of the thread. We specifically do this in one designated thread to avoid

Algorithm 18: SEER algorithm.

```
41: ACQUIRE-SEER-LOCKS(thread, txId, htmStatus)
42:   if htmStatus & _XABORT_CAPACITY ∧ ¬thread.acquiredCoreLock
43:     ▷ adapted to the topology of hyper-threads in Intel processors
44:     acquire-lock(coreLocks[thread.core % PHYSICAL_CORES])
45:     thread.acquiredCoreLock ← true
46:   if attempts = 1
47:     ACQUIRE-TX-LOCKS(txId)                    ▷ acquire locks in row locksToAcquire[txId]
48:     thread.acquiredTxLocks ← true

49: WAIT-SEER-LOCKS(thread, txId)
50:   if is-locked(sgl)                  ▷ avoid starting hardware transactions if the fall-back is in use
51:     if thread.core = 0                          ▷ only one thread updates the serialization locks
52:       UPDATE-SEER-LOCKS()                        ▷ exploit the wait time to run SEER
53:       if enough-samples()
54:         stochastic-hill-climbing(Th1, Th2)                ▷ periodically adapt the parameters
55:     wait while is-locked(sgl)                  ▷ wait here instead of aborting in line 12
56:   ▷ if some other thread is owning these SEER locks, cooperate with it and wait
57:   wait while ¬thread.acquiredTxLocks ∧ is-locked(txLocks[txId])
58:   wait while ¬thread.acquiredCoreLock ∧ is-locked(coreLocks[thread.core])

59: RELEASE-SEER-LOCKS(thread, txId)
60:   if thread.acquiredTxLocks
61:     RELEASE-TX-LOCKS(txId)
62:   if thread.acquiredCoreLock
63:     release-lock(coreLocks[thread.core])
```

synchronization. Furthermore, we have an active transactions list with as many slots as threads in the program, making each entry of the list a single-writer multi-reader register.

The procedure to acquire the transaction locks simply goes over the row *locksToAcquire[txId]* and acquires each lock. All rows are sorted consistently by the periodic update, hence this procedure acquires them in that order to avoid deadlocks. We also optimize this procedure to acquire the locks with a hardware transaction when there are two or more locks, instead of performing multiple compare-and-swap operations (CAS) to acquire all locks. The rationale of this optimization is to batch the synchronization of two or more CASes into a single hardware transaction. If the transaction is not successful, we fall-back to the normal acquisition. Note that this is not lock elision [Rajwar and Goodman, 2001]; we are effectively using HTM as a multi-CAS, not eliding the locks acquired.

Devising the Locking Scheme. We are left with the logic for updating the locking scheme for fine-grained serialization of transactions in SEER, which we present in Algorithm 19. This procedure, opportunistically invoked by the designated thread, while waiting for the single-

Algorithm 19: SEER algorithm.

```
64: UPDATE-SEER-LOCKS()
65:   for all x ∈ A                                  ▷ A is the set of txs in the application source code
66:     η ← average({P(x aborts ∩ x‖y), ∀y∈ A})
67:     σ² ← variance({P(x aborts ∩ x‖y), ∀y∈ A})
68:     for all y ∈ A                                ▷ determine if y is likely to contend with x
69:       ▷ 1st condition checks whether abort events of x, in which y is seen running concurrently, are
          common enough
70:       ▷ 2nd condition checks if y is among the txs that, when executed concurrently with x, most
          likely contend with x
71:       if ( P(x aborts ∩ x‖y) > Th1 ∧
            P(x aborts ∩ x‖y) > Th2-th percentile of a Gaussian N(η,σ²) ) then
72:         locksToAcquire[x] ← y                  ▷ contending txs take each other's locks upon abort
73:         locksToAcquire[y] ← x                  ▷ txs also wait for their tx locks to be free (line 57)
74:   ▷ sort all locks in each row of locksToAcquire, and swap the old matrix by the new one (using an
      indirection pointer)
```

global lock to be released, starts by aggregating the commit and abort statistics gathered on a per-thread basis. The designated thread sweeps over the statistics of the other threads without any synchronization explicitly enforced, thus being subject to racy accesses. We mitigate the hazard of "out of the blue" values by placing the data in word-aligned memory locations.

For each pair of transactions, x, y, of the application, we calculate the conjunctive probability of x aborting *and* y running concurrently (i.e., $\mathcal{P}^{conj}_{x,y} = P(\text{x aborts} \cap x \parallel y)$):

$$\mathcal{P}^{conj}_{x,y} = P(\text{x aborts} \cap x \parallel y) = P(\text{x aborts} \mid x \parallel y) \times P(x \parallel y)) = \frac{a_{x,y}}{c_{x,y} + a_{x,y}} \times \frac{c_{x,y} + a_{x,y}}{\sum_{k=0}^{|A|} a_{x,k} + c_{x,k}} = \frac{a_{x,y}}{\sum_{k=0}^{|A|} a_{x,k} + c_{x,k}}$$

where we abbreviated the elements of the matrices commitStats$[x][y]$ as $c_{x,y}$ and abortStats$[x][y]$ as $a_{x,y}$. Note that this probability calculation can be efficiently approximated with the statistics that are at our disposal, as shown in the previous algorithms, where we describe how to collect them.

As shown in Algorithm 19, we must then identify the probabilities that are most meaningful, and for which locking should pay off in terms of forbidding some parallelism but avoiding significant hardware transaction aborts (and thus diminish the reliance on the single-global lock). To achieve this, we resort to two thresholds, $\mathcal{T}h_1$ and $\mathcal{T}h_2$, which are aimed at pursuing different goals.

The threshold $\mathcal{T}h_1$ establishes a lower bound on $\mathcal{P}_{x,y}^{conj}$, below whose value SEER avoids serializing transactions x and y. Low values of this probability imply that the frequency of aborts events of x, in which y was found to run concurrently with it, are rare. It is thus beneficial to avoid the cost of restricting concurrency and sparing the costs of additional lock acquisitions.

The threshold $\mathcal{T}h_2$ is instead used to establish a cut-off on the probability distribution of $\mathcal{P}_{x,y}^{conj}$, which aims at determining a subset $\mathcal{S}$ among the transactions available in A that are responsible for the most likely conflicts with a given x.

More in detail, SEER includes in $\mathcal{S}$ only each transaction y whose $\mathcal{P}_{x,y}^{conj}$ is larger than the $\mathcal{T}h_2$-th percentile of a Gaussian distribution $\mathcal{N}(\eta, \sigma^2)$ with mean η and variance σ equal, respectively, to the mean and variance of the values of $\mathcal{P}_{x,y}^{conj}$ (for each transaction y with respect to a given x).

In other words, this second threshold aims at identifying the transactions y that have the *relatively* highest probability values of conflict with x. This contrasts with the first threshold, which instead identifies the transactions that have the largest *absolute* values of conflict with x.

SEER can overestimate that *absolute* probability threshold for two main reasons. First, and most importantly, SEER can falsely blame a transaction y for having aborted x, merely because y is spotted as executing right after x gets aborted. Second, in order to minimize monitoring overheads, SEER adopts a lightweight and inherently imprecise mechanism for tracking concurrency among transactions, which also causes falsely blamed transactions.

As we shall show in Section 6.6, due to this risk of overestimation, if SEER were to use only $\mathcal{T}h_1$, it could overly restrict parallelism for certain workload types (e.g., when there are few transaction types and a high number of concurrent threads). By using also $\mathcal{T}h_2$, though, SEER can reason on the distribution of conflict probabilities and pinpoint in a more accurate way which transactions are actually the most likely causes of conflict, which can be beneficial to reduce false positive rates of locks.

Summarizing: if both conditions in line 71 are met, meaning that x is deemed to abort too often because of y, SEER requires that transactions x and y have to acquire each other's lock (recall that we associate one lock per transaction).

Finally, SEER relies on an on-line self-tuning mechanism that automates the setting of the values of the thresholds $\mathcal{T}h_1$ and $\mathcal{T}h_2$, hence sparing users from the burden of identifying stati-

cally defined values that may be sub-optimal in heterogeneous, or time varying, workloads. To this end, SEER uses a simple and lightweight bi-dimensional stochastic hill climbing search, which exploits the feedback of the TM performance (throughput obtained via RTDSC-based measurements) to guide the search in the parameter's space [0,1]×[0,1] for the thresholds $\mathcal{T}h_1$ and $\mathcal{T}h_2$. Our hill climbing is stochastic in the sense that, with a small probability p, it performs random jumps in the parameters' space to avoid getting stuck in local minima. We configured this self-tuning mechanism with standard values that were applied to irregular concurrent applications such as those used with TM [Diegues and Romano, 2015a]. Specifically, we set p to 0.1% and the initial values of $\mathcal{T}h_1 = 0.3$ and $\mathcal{T}h_2 = 0.8$.

6.6 Validating the Design Choices of SEER

In order to motivate the choices made while designing SEER, we first validate the proposed probabilistic lock inference scheme by evaluating its accuracy via a simulation study. This choice allows us to assess the accuracy of SEER when faced with a large number of randomly generated, yet known *a priori*, synthetic workloads. Further, by relying on a simulator, we can exert tight control on the degree of accuracy with which SEER can assess concurrency among transactions. This allows us to study the impact on SEER's locking inference scheme due to imprecise information regarding concurrently executing transactions.

In the validations that we shall conduct in this section, we always synthesize the conflict likelihood between transactions, so that they are known a priori and enforced in the execution by our simulator.

Specifically, in the simulation model we use a *conflict matrix* $\mathcal{C}$, whose cells $\mathcal{C}_{i,j}$ define the probability for transaction Tx_i to be aborted by transaction Tx_j, in case they run concurrently.

Table 6.3: Simple example for a matrix $\mathcal{C}$ with 4 types of transactions where their conflicts are known *a priori*

-	1	2	3	4
1	**1.00**	0.00	0.00	0.10
2	0.00	0.20	0.00	**0.50**
3	0.00	0.01	0.05	0.00
4	**1.00**	0.00	0.00	0.00

Table 6.4: Description of parameters used in the simulations.

Parameter of the configuration	Values used
number of simulations per configuration	100 000
number of rounds per simulation	100 000
number of threads	uniformly distributed in $[2; 32]$
number of transaction types	uniformly distributed in $[2; 32]$
chance of running a transaction	75%
chance of choosing a specific transaction type	chosen given uniform distribution
transaction conflicts	given by $\mathcal{C}$
Zipfian distribution for $\mathcal{C}$	given by $\mathcal{Z} \in [1.0; 2.5]$

An example of matrix $\mathcal{C}$ is shown in Table 6.3: in this case Tx_4 aborts deterministically (i.e., probability of 100%) if it runs in concurrency with transaction Tx_1, whereas transaction Tx_1 aborts with probability 10% if it executes concurrently with Tx_4.

Table 6.4 summarizes the key parameters used in the simulation model used in this study. Each data point in the following plots corresponds to the average of 100 000 simulations. In each simulation, we vary the number of threads, transaction types, and the conflict matrix as indicated in Table 6.4. In particular, each row of the conflict matrix is generated using a Zipfian distribution, whose $\mathcal{Z}$ parameter is treated as an independent variable in our study. In the following, we shall vary $\mathcal{Z}$ in the $[1.0; 2.5]$ range, where we recall that $\mathcal{Z}$ values closer to 1.0 generate more uniform distributions, and larger values result in more skewed distributions. This allows to mimic realistic cases (such as those we test later) where there are large discrepancies between the conflict patterns of different transaction types.

Each simulation consists of the execution of 100 000 *rounds*. In each round, each thread first chooses whether to run a transaction or not (given a likelihood of executing transactional work of 75%). In the positive case, it picks one of the transactions available in the workload with equal probability. In each round, for each pair of threads, it is decided whether a conflict occurs between the transactions that they are executing based on the conflict matrix $\mathcal{C}$. If a transaction Tx_i conflicts, with probabilities according to $\mathcal{C}$, with at least one other transaction active in a different thread, then Tx_i is aborted.

Note that we simulate also the usual retry policy for the transactions, which may dictate the usage of the fall-back path using a single-global lock. Each thread concludes the round by looking at the *activeTxs* and updating its thread-level statistics accordingly. This lock-step round-based

procedure provides exact information on which other transactions were concurrently active during an execution round. Later, in this section, we extend this simulation model to include scenarios in which the sampling of the *activeTxs* array can lead to obtaining erroneous information. This will allow us to simulate more closely SEER, which, we recall, relies on minimal synchronization and possibly inaccurate information.

We highlight that this simulation model encompasses a representation also of the fall-back path with a single-global lock: hardware transactions resort to a single-global lock, after attempting to use HTM 5 times, thus forbidding all concurrent transactions from being executed in that round.

It should be noted that our simulation model does not encompass spurious aborts triggered by the HTM. Also, our simulation model does not capture the effects of running multiple hardware threads on the same physical core. Hence, this study does not aim to evaluate the effectiveness of core locks - for which the reader should refer to Section 6.7.

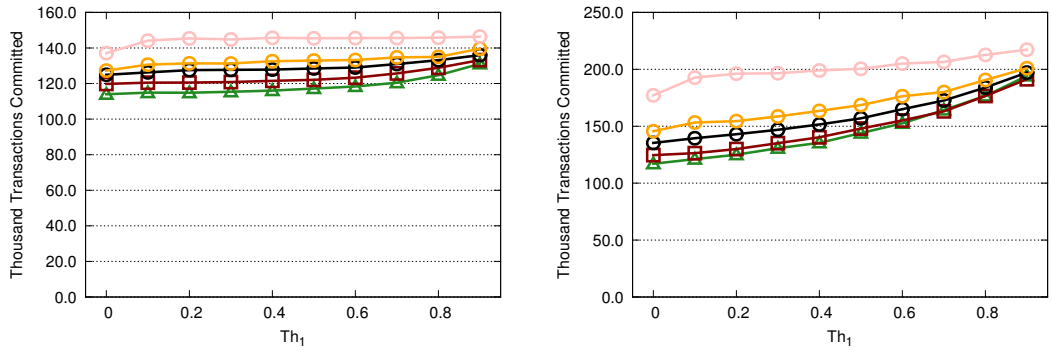
In the simulator, we monitor the execution of the first third of the rounds without acquiring any SEER-induced lock. On the basis of the statistics gathered during this preliminary phase, we use the same logic of SEER to compute $\mathcal{P}_{x,y}^{conj}$.

We then execute the remaining two thirds of the 100 000 rounds, using the locks chosen by SEER, and evaluate its accuracy as follows:

$$accuracy = \frac{truePositives + trueNegatives}{totalEvents} \tag{6.1}$$

A true positive happens when there would exist a conflict between a pair of transactions that is prevented by a lock devised by SEER. In contrast, a true negative corresponds to a pair of transactions that executes concurrently without conflicting, and for which SEER did not decide to enforce a transaction lock. The total number of events is the number of rounds — in each of which we execute this logic — times the number of pairs of transactions that execute concurrently.

Note that, while SEER relies on a throughput driven hill climbing-based self-tuning procedure to set the values of the $\mathcal{T}h_1$ and $\mathcal{T}h_2$ thresholds, in this simulation study we shall treat these



(a) Conflict matrices $\mathcal{C}$ generated for each simulation from Zipfian distribution with $\mathcal{Z} = 1.0$.

(b) Conflict matrices $\mathcal{C}$ for each execution simulation from Zipfian distribution with $\mathcal{Z} = 1.5$.

(c) Conflict matrices $\mathcal{C}$ for each execution simulation from Zipfian distribution with $\mathcal{Z} = 2.0$.

(d) Conflict matrices $\mathcal{C}$ for each execution simulation from Zipfian distribution with $\mathcal{Z} = 2.5$.

Figure 6.3: Average accuracy for SEER in 4 scenarios varying the parameter $\mathcal{Z}$ of the Zipfian distribution that is used to generate the conflict matrices $\mathcal{C}$ used as input for each simulation. In each plot we vary the value of both thresholds that are applied to $\mathcal{P}^{conj}_{x,y}$.

thresholds as two independent variables. This choice allows us to gain additional insights on the impact of tuning these two thresholds independently. We will evaluate the gains achievable by using SEER's self-tuning hill climbing strategy versus static thresholds in Section 6.7, when presenting the experimental evaluation of our prototype.

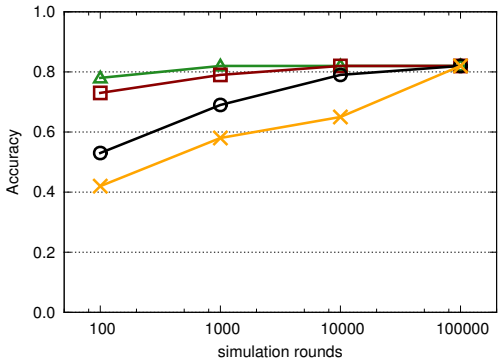
We present in Figure 6.3 the average accuracy in 4 scenarios with different parameters for the Zipfian distribution that generates the conflict matrices $\mathcal{C}$. In each plot, we show the results for different values of both thresholds that are applied to $\mathcal{P}^{conj}_{x,y}$.

In general, there are two relevant trends that emerge from these data: using the value of 0.0, for either $\mathcal{T}h_1$ or $\mathcal{T}h_2$, results in worse performance than if both thresholds are set to some, properly selected, positive value. As such, this suggests that the proposed approach of combining these thresholds results in better accuracy in terms of locks chosen, which matches our earlier intuitions.

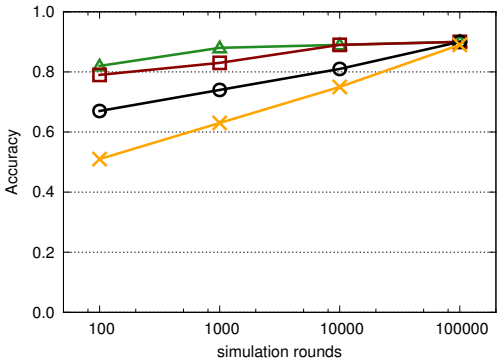
We also note that the correct settings of the thresholds appear to have a smaller impact for the scenarios in which the conflict matrices are more skewed. Further, the relevance of properly setting the values of the $\mathcal{T}h_1$ and $\mathcal{T}h_2$ thresholds is higher in more skewed workload scenarios.

This can be explained by considering that, the more skewed the conflict matrix $\mathcal{C}$, the smaller the set of transactions that cause most of the aborts for a given a transaction Tx. This has the effect of reducing the uncertainty that SEER 's lock inference scheme has to cope with. In fact, SEER's design is tailored to identify a small subset of the conflicting transactions — the one that matters the most — exactly because skewed workloads are frequently found in realistic applications [Cooper et al., 2010] (namely, of those that we evaluate also later). That is, normally, not every transaction conflicts often with every other transaction type.

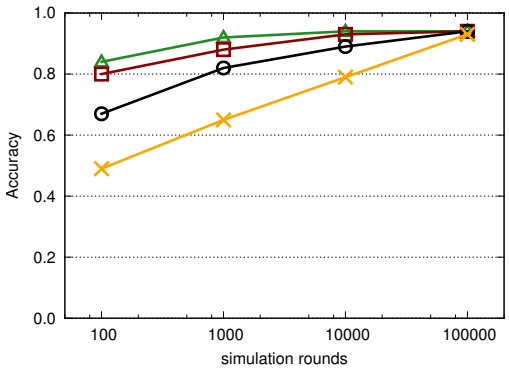
Another important trend is visible across the plots: since the workloads have different characteristics, the best accuracy is achieved when using different combinations of the threshold values. For more uniform workloads, e.g., Figure 6.3(a), it is important to use a high $\mathcal{T}h_2$ value, since with uniform workloads the conflict probabilities tend to be very similar. Thus, the use of high values for $\mathcal{T}h_2$ allows for identifying more easily the top conflicting transactions among those that are already above the average for a given transaction. When the workloads are more skewed, in contrast, it is more important to use a sufficiently high $\mathcal{T}h_1$ value that allows to distinguish the high conflicts from the lower ones, in order to achieve higher accuracy levels.



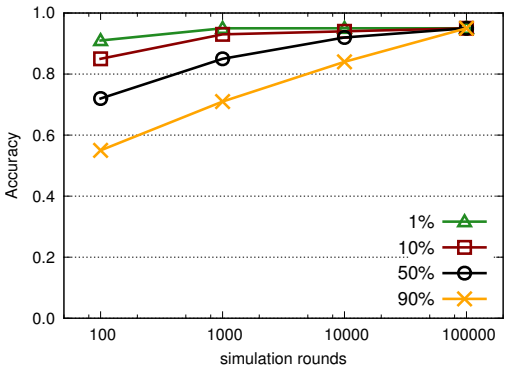
(a) Conflict matrices $\mathcal{C}$ generated for each simulation from Zipfian distribution with $\mathcal{Z} = 1.0$.



(b) Conflict matrices $\mathcal{C}$ for each execution simulation from Zipfian distribution with $\mathcal{Z} = 1.5$.



(c) Conflict matrices $\mathcal{C}$ for each execution simulation from Zipfian distribution with $\mathcal{Z} = 2.0$.



(d) Conflict matrices $\mathcal{C}$ for each execution simulation from Zipfian distribution with $\mathcal{Z} = 2.5$.

Figure 6.4: Average accuracy, when varying the number of simulation rounds, to show how the convergence time is affected by the imprecise gathering of statistics when accessing the *activeTxs* list. In each plot, we show four alternatives that simulate different probability values (p_{err}) of sampling a random concurrent transaction instead of the correct one. We show four scenarios, similarly to Figure 6.3, by varying the parameter $\mathcal{Z}$ of the Zipfian distribution that is used to generate the conflict matrices $\mathcal{C}$ used as input for each simulation. In each plot, we use the thresholds for $\mathcal{T}h_1$ and $\mathcal{T}h_2$ that provided the best accuracy in the corresponding scenario in Figure 6.3.

Overall, these results suggest that the optimal settings of the thresholds $\mathcal{T}h_1$ and $\mathcal{T}h_2$ is largely workload dependent: the difference in accuracy can vary from 22% (in Figure 6.3(a)) to 48% (in Figure 6.3(d)). These experimental data motivate the use of SEER's hill-climbing based self-tuning approach, rather than the use of static settings.

In general, these results are also important as they attest the high accuracy that is achievable by SEER in a wide range of workloads, i.e., 90% average accuracy across all the workloads with proper threshold tuning.

Finally, we analyze also the impact of feeding the lock inference scheme of Seer with approximate/imprecise information regarding the set of concurrently active transactions. Recall that, in the experiments analyzed so far, concurrent transactions have always been inferred correctly, as if there was a global synchronization point when accessing the *activeTxs* list.

The plots shown in Figure 6.4, instead, allow us to study how the accuracy of Seer changes when varying the probability of feeding it with imprecise information regarding concurrent transactions. In this case, whenever a transaction Tx samples an entry of the *activeTxs*, it obtains a random transaction type, instead of the correct one, with probability p_{err}. This allows us to simulate the real behaviour of SEER, which samples transactions from the *activeTxs* list without any synchronization, and is thus vulnerable to erroneously consider as concurrent transactions that had either already completed when Tx aborted, or that had not yet started when Tx aborted.

The plots in Figure 6.4 consider the same workload settings of Figure 6.3, and use the values of $\mathcal{T}h_1$ and $\mathcal{T}h_2$ that yielded the best accuracy in the previous experiments. In this case, though, on the horizontal axis, we let the number of simulation rounds vary from 100 to 100 000.

The reason for this is that, by increasing the probability p_{err} of wrongly inferring concurrency, we increase also the noise in the data collected by SEER. As the plots clearly show, this has only the effect of delaying the correct inference of the actual conflict probabilities, as with a sufficiently large number of observations the accuracy converges to the values observed in Figure 6.3.

Furthermore, it is also evident that, when gathering statistics for a low number of rounds, it is more likely to infer concurrent transactions wrongly. However, as we increase the number of rounds, the accuracy quickly grows up to expected value. We also highlight that, in TM applications it is easily the case that throughput rates are well over thousands transactions per second. This is easy to understand given that transactions are typically small and execute over

in-memory data. As such, these numbers of simulated rounds correspond to a very short real time period. Hence, in practice, SEER can infer a highly accurate locking scheme in a robust and timely way.

6.7 Experimental Comparison with Other Systems

We now report the results of an experimental study based on a fully fledged prototype of SEER, in which we compare the proposed solution with three other state of the art systems.

To evaluate our proposal, we formulate several questions and design a set of experiments aimed to answer them:

- *What are the gains achievable by* SEER*?* In Section 6.7.1, we compare SEER with three alternative techniques for regulating the execution of HTM transactions.
- *How are those gains obtained?* In Section 6.7.2, we provide detailed data on how often hardware transactions are aborted and why.
- *How are transactions scheduled?* In Section 6.7.3, we assess how often hardware transactions are successful and to what extent the various available locks are acquired.
- *What are the overheads of* SEER*?* In Section 6.7.4, we assess the performance slowdown of running SEER's monitoring and lock inference mechanisms, but without using the locks synthesized by SEER.
- *What are the relative merits of each component of* SEER*'s architecture?* In Section 6.7.5 we create and compare several variants of SEER that incrementally use the different modules of the proposed system.

All the results in this chapter were obtained using one of the largest and most recent Intel processors with HTM support, whose description is available on Table 6.5, and of which we highlight the fact that it has 28 virtual cores (14 physical cores, each one running up to 2 hardware threads with Intel Hyper-Threading). The machine was equipped with 128GB of RAM and ran Ubuntu 14.10. The results reported in this chapter are the average of 10 runs.

Table 6.5: Characteristics of the new Haswell machine used to evaluate SEER with HTM support and 28 cores.

Resource	Description
Processor	Xeon E5-2683 v3 2.0GHz
Cores	14 (each with hyper-threading)
L1 Cache	32KB 8-way (per core)
L2 Cache	256KB 8-way (per core)
L3 Cache	35MB (shared)
Cache Line	64B
RAM Size	128GB
Operating System	Ubuntu 14.10

Our evaluation uses the standard STAMP suite [Minh et al., 2008], a popular set of benchmarks for TM, encompassing applications representative of various domains that generate heterogeneous workload. We excluded Bayes given its non-deterministic executions, and Labyrinth as most of its transactions exceed Intel HTM capacity. In addition to these, we have also used the STMBench7 benchmark [Guerraoui et al., 2007], which is particularly interesting for scheduling as it encompasses 45 different transactions types. Due to the limitations of Intel HTM capacity, we have reduced the working-set of STMBench7's workloads roughly by ten times (namely, the number of components per module to 1 and levels of assemblies to 2) so that there is a relatively small number of aborted transactions due to capacity (less than 20%, in contrast with over 50% normally).

6.7.1 How Much Can We Gain With Seer?

To assess the benefits of SEER we compare it with three alternatives for the concurrency control:

- **HLE** where transactions may be retried a small number of times (processor implementation-dependent), but without any scheduling or contention management. As pointed out in the literature [Dice et al., 2008] this can cause a lemming effect on the elided lock, i.e., failed hardware transactions keep exhausting the attempts and fall-back to using the single-global lock [2].

[2] All the benchmarks herein used have a transactional interface, i.e., were written using atomic blocks and, as such, there is only one (global) lock to elide (maintained internally by the TM library).

- **Baseline** where the retry logic is controlled in software: we retry a given number of attempts in HTM and always wait before doing so if the single-global lock is taken. As already discussed in Section 6.3, the usage of a single lock in the fall-back path of these two baseline mechanisms makes them analogous in spirit to the ATS scheduler [Yoo and Lee, 2008].

- **SCM** where we implemented the Software-assisted Conflict Management [Afek et al., 2014] technique. SCM uses an auxiliary lock to serialize transactions that are aborted, thus decreasing the chance of having the lemming effect.

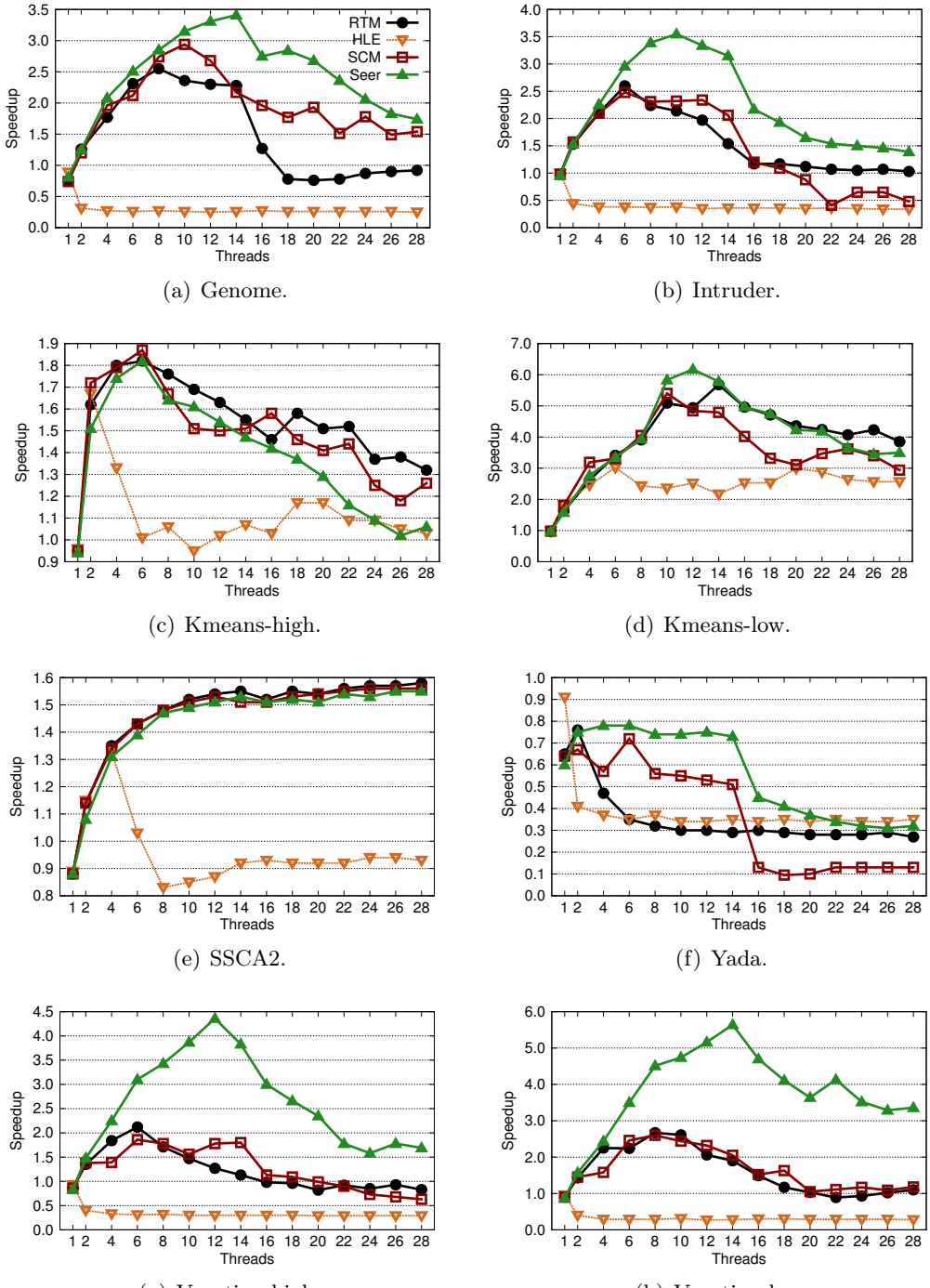
We used a budget of 5 attempts for hardware transactions RTM, SCM and Seer, which is also the value used by Intel researchers for a similar set of benchmarks [Yoo et al., 2013].

We first present the results for this comparison with STAMP in Figure 6.5. The speedups are relative to a sequential non-instrumented execution. In general, we can see that Seer performs better or similar to the best alternative. This best alternative is some times SCM, but other times Baseline, with some advantage of the former when there are less threads used. Finally, HLE performs significantly worse than the others.

By analyzing these results with focus on Seer, we can identify three groups of benchmarks with different merits of performance:

- Genome, Intruder, Vacation-high and Vacation-low: these benchmarks clearly illustrate the advantage of the proposed probabilistic scheduler for hardware transactions. The gains of Seer are consistent and unveil a new scalability ceiling with respect to the baselines using the same hardware parallelism. Furthermore, when exploiting the highest number of threads available, the scheduling devised by Seer minimizes performance degradation and avoids trashing.

- SSCA2, Kmeans-low, Yada: performance is generally on par with the best of the considered alternatives. As we shall see in more detail in the rest of the evaluation, this happens because: 1) the contention level is very limited, meaning there are almost no conflicts to avoid (SSCA2); or 2) transactions are small and the workload is not 100% transactional, for which reason reliance on the single-global lock is not very harmful (Kmeans-low); or 3) reducing conflicts results in more capacity aborts (Yada).



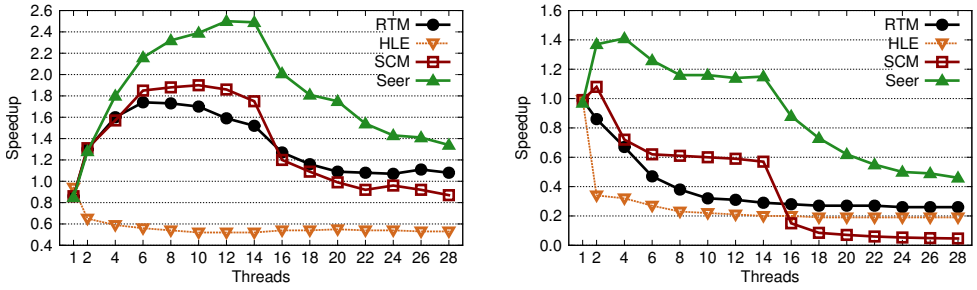
(a) Genome. (b) Intruder.

(c) Kmeans-high. (d) Kmeans-low.

(e) SSCA2. (f) Yada.

(g) Vacation-high. (h) Vacation-low.

Figure 6.5: Speedup of different HTM based approaches across **STAMP** benchmarks.



(a) Geometric mean for STAMP.

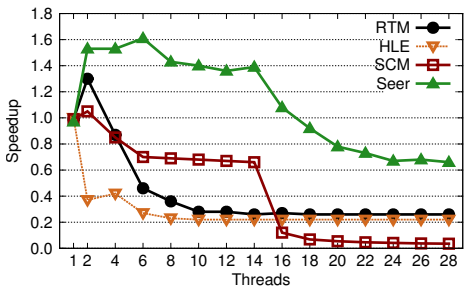
(b) Geometric mean for STMBench7.

Figure 6.6: Geometric mean speedup for STAMP and STMBench7.

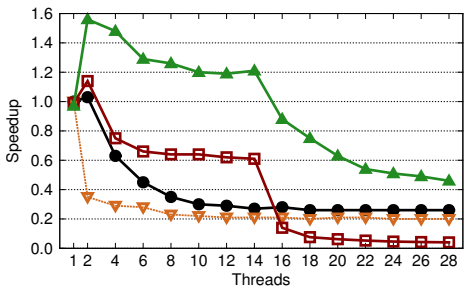
- Kmeans-high: in this case the performance of SEER is actually decreasing (with respect to the best alternative) as the number of thread increases. This happens because transactions in this workload, although quite conflicting, are still very short. As a result, the locking scheme devised by SEER does not pay off because, when using the baselines alternatives instead, the transactions that abort take the single-global lock for such a short duration that it has only a limited impact on performance. Ideally, SEER would eventually self-tune its thresholds to prevent this situation, as it happens in some of the other benchmarks where its performance is similar to the baselines. However, for this benchmark, we have confirmed that this does not happen because of the short running time of this benchmark (lasting less than a second for most of the parallel executions). To verify this, we have experimentally executed the benchmark in a closed loop, while preserving the statistics collected by SEER, and verified that it converges to the performance of the other baselines (namely, of Baseline) in 3 to 5 runs.

Finally, with respect to STAMP, we analyze also the geometric mean speedup across all benchmarks that is shown in Figure 6.6(a). This data evidences that, despite some workloads where performance remains on par with some of the alternatives, the gains are quite substantial for most of the considered number of threads: namely, 64% over Baseline and 42% over SCM at 14 threads (i.e., the peak of average speedup); and up to 77% over SCM at 20 threads. The gains are also up to 4.8× over HLE.

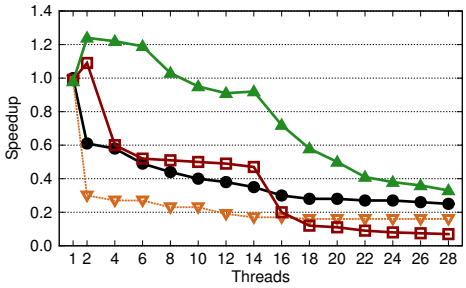
Moving on to STMBench7 [Guerraoui et al., 2007], we present results for three workloads, in which we vary the the percentage of read-only transactions. This is a particularly challenging setting for a TM system because the 45 different transaction types have very heterogeneous characteristics, ranging from very small to huge transactions, small and large read working sets,



(a) Read-dominated.



(b) Read-write balanced.



(c) Write-dominated.

Figure 6.7: Speedup of different HTM based approaches across **STMBench7** workloads.

and varying contention degrees. As such, this opens more opportunities in general for scheduling, and results also in a more complex test for SEER.

It is also noteworthy that these workloads are quite contended, as it is visible by the rather small scalability potential and speedup values; this stems not only from the nature of the benchmark, as evidenced by other evaluations with Software TMs [Fernandes and Cachopo, 2011, Dragojević et al., 2009a, Rito and Cachopo, 2014], but also from the working-set reduction that we performed to make adequate it to the capacity of hardware transactions.

Although the absolute speedups vary with the workload changes, the relative merits of each solution remain approximately the same. As such, we focus on the geometric mean across the three workloads, which is shown in Figure 6.6(b). There, we observe an improvement of SEER of at least 2× at 4 threads (i.e., the peak of average speedup) and up to 3.7× over Baseline and 5.3× over SCM at 14 threads.

The trend across all benchmarks is quite visible and solid: SEER enables more scalability up to some degree of parallelism and then avoids drastic performance plunges when the contention becomes very high with dozens of concurrent threads.

6.7.2 Where Are the Gains of Seer Coming From?

The approach taken by Seer is to bridle parallelism so that transactions can make a more efficient use of HTM.

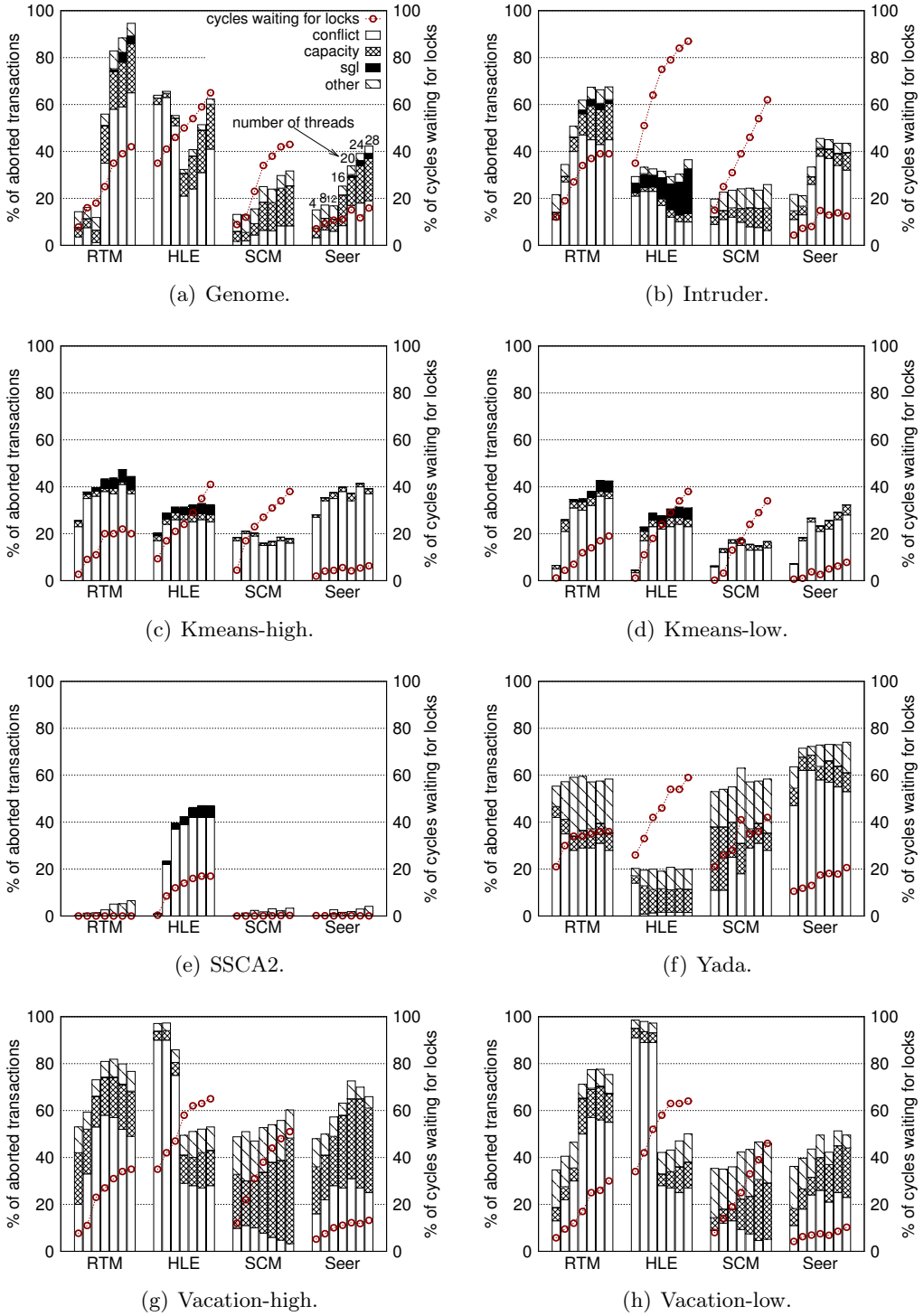
In the previous subsection we already quantified the performance benefits globally achievable by Seer. We now focus our experimental study to shed light on the origins of the speedups that it achieves. In particular, we aim to answer the following two key questions: 1) to what extent aborts of hardware transactions are reduced; and 2) to evaluate the consequent reduction of activation of the fall-back path.

Once again, we show results for both the STAMP (in Figure 6.8) and STMBench7 (in Figure 6.9) benchmarks. In those plots we can see, on the one hand, the percentage of hardware transaction aborts (specified by type) and, on the other hand, the percentage of processor cycles spent waiting for the single-global lock to become available. For SCM, we include in this processor cycles count also the cycles spent spinning on the auxiliary lock. For Seer we also report the cumulative percentage of cycles spent waiting for transactions and core locks. This data is shown for each workload and while varying the number of threads for both Seer and for the three considered baselines.

We distinguish three types of aborts for hardware transactions:

- *conflict*: aborts due to concurrent threads issuing contending data accesses. Note that the subscription of the single-global lock may generate aborts that are reported as conflicts: when a hardware transaction starts, it verifies that the single-global lock is free and continues to execute the transaction, so that if the lock gets acquired the hardware transaction shall abort.
- *capacity*: aborts triggered when a hardware transaction accesses more cache lines than those that fit the hardware limits (namely, of the processor caches).
- *sgl*: explicit abort requests triggered in case a hardware transaction finds the fall-back lock busy upon having subscribed it.
- *other*: aborts triggered due to other reasons, e.g., page faults and interrupts.

In general, more than half of the aborts are due to conflicts. These are the ones that feed



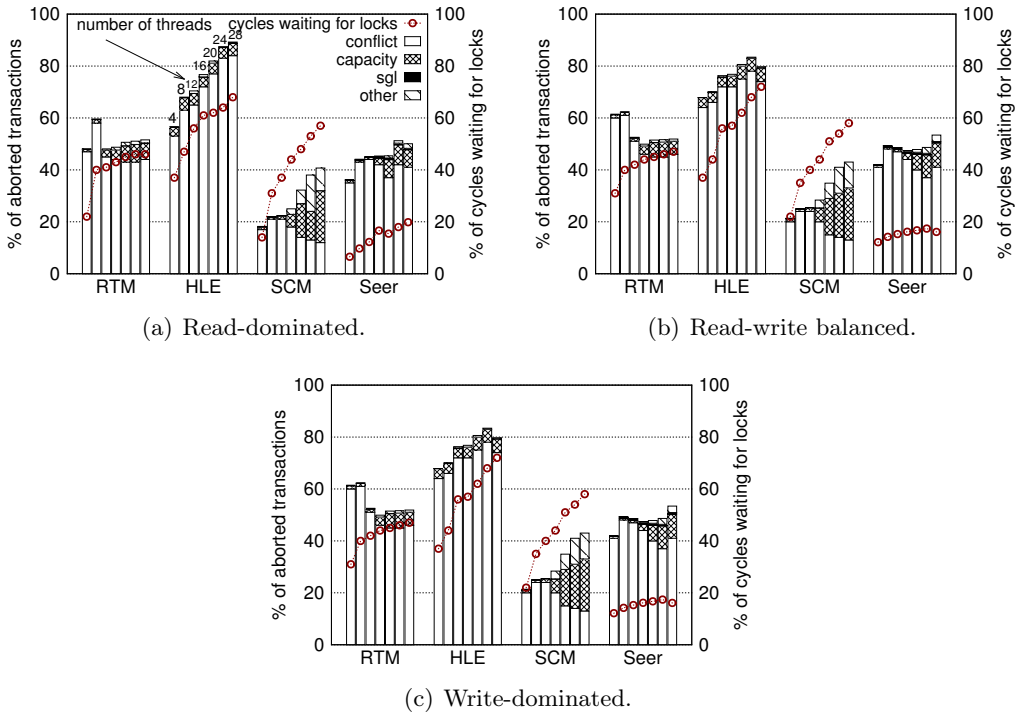
(a) Genome. (b) Intruder.

(c) Kmeans-high. (d) Kmeans-low.

(e) SSCA2. (f) Yada.

(g) Vacation-high. (h) Vacation-low.

Figure 6.8: Number and types of aborts suffered by hardware transactions across the **STAMP** benchmarks, together with the percentage of processor cycles spent waiting for the single-global lock to be available (including the auxiliary lock for SCM). In the case of Seer we include also the cumulative percentage of cycles spent spinning on transaction and core locks.



(a) Read-dominated.

(b) Read-write balanced.

(c) Write-dominated.

Figure 6.9: Similar to Figure 6.8, but instead for the **STMBench7** workloads.

SEER's statistics, and that the transaction locks try to reduce. Then, most of the remaining aborts are due to capacity, which SEER attempts to limit via the core locks.

By analyzing each benchmark, we can see that SEER does reduce the total aborts in several cases (namely in the Genome, Intruder, Vacation and STMBench7 benchmarks) when compared to the approach with Baseline. There are two noteworthy remarks with respect to the other two alternatives. Firstly, recall that HLE insists very little on hardware transactions, and so it ends up aborting less in total because it is the only approach that does not have a budget of 5 retries per transaction (as experimental evaluation seems to indicate that it retries 1 or up to 2 times at most). Secondly, SCM does have the same budget of 5 retries as SEER, but since it serializes aborted transactions via the auxiliary lock, it restricts parallelism upfront and in a coarse fashion; recall that SCM relies on a single auxiliary lock, whereas SEER's locks are fine grained.

As such, these two alternatives manage to reduce transaction aborts. Yet, this comes at the cost of severely hindering parallelism. An evidence of this claim can be clearly obtained by analyzing the data regarding the processor cycles spent waiting for the fall-back lock (and, for SCM, also for the auxiliary lock) to be free. For the case of SEER we show not just the

cycles spent waiting for the global lock, but also for Seer's transaction and core locks. When considering the average cycles spent waiting for locks, we have that Seer has less 4× than HLE, 3.5× than SCM, and 3× than Baseline. Notice that, while Baseline has the closest cycles spent waiting comparing to Seer, it is also the alternative that consistently aborts more. This tension between waiting for locks and aborting transactions is a trade-off that is best managed by Seer's ability to generate fine-grained locks that prevent aborts while at the same time avoiding too much waiting.

Another interesting aspect to highlight is that, in some benchmarks, the ability of Seer to reduce conflict aborts leads to an increase of capacity aborts. This can be explained by considering that transactions whose conflicts are avoided, thanks to Seer's transactions locks, now advance longer in their execution and hence become more vulnerable to capacity exceptions. We observed this phenomenon experimentally, for instance in Vacation-high. In this case, we can see that the transaction locks of Seer reduce the conflicts (when comparing to Baseline) but then Seer ends up with more capacity aborts — which the core locks do not amortize entirely. Baseline, on the other hand, does not observe those capacity aborts because the hardware transactions abort early due to conflicts.

6.7.3 How are Transactions Being Scheduled?

To understand the reasons that lead Seer to reduce aborts and wait time for the fall-back lock, we now present results that explain how transactions are being scheduled. Namely, we seek to understand how often transactions rely on the single-global lock (thus giving up on the HTM support), or when they acquire other locks (in the case of SCM and Seer).

Table 6.6 provides a breakdown of the usage of locks for each considered approach, providing additional insights on the reasons underlying the performance gains achieved by Seer. We report data for 14 and 28 threads, for each benchmark and workload. To interpret this data, it is important to consider that, for each approach, it is desirable that the top row(s) have the highest frequency, because they are the ones that constrain parallelism the least, while still executing with HTM support successfully.

By considering Genome, we can see that HLE has a considerable portion of transactions executing under the single-global lock, due to its fragile retry on abort policy. This is a trend

Table 6.6: Breakdown of percentage (%) of transaction execution modes when running with 14 and 28 threads.

		Genome		*Intruder*		*Kmeans-H*		*Kmeans-L*		*SSCA2*	
Variant	Execution Mode	14t	28t	14t	28t	14t	28t	14t	28t	14t	28t
HLE	HTM no locks	71	71	77	77	70	70	71	72	76	78
	SGL fall-back	29	29	23	23	30	30	29	28	24	22
Baseline	HTM no locks	97	90	92	89	76	76	89	82	100	100
	SGL fall-back	3	10	8	11	24	24	11	18	0	0
SCM	HTM no locks	91	82	84	72	76	74	75	75	98	98
	HTM + Aux lock	7	16	14	24	23	25	24	24	2	2
	SGL fall-back	2	2	2	4	1	1	1	1	0	0
Seer	HTM no locks	96	79	87	76	75	59	90	76	100	100
	HTM + TxLocks	2	3	5	11	8	24	5	14	0	0
	HTM + CoreLocks	0	4	0	2	0	0	0	0	0	0
	HTM + Tx + CoreLocks	0	8	0	2	0	1	0	1	0	0
	SGL fall-back	2	6	8	9	17	16	5	9	0	0

		Yada		*Vacation-H*		*Vacation-L*		*STMB7-R*		*STMB7-RW*		*STMB7-W*	
Variant	Execution Mode	14t	28t	14t	28t	14t	28t	14t	28t	14t	28t	14t	28t
HLE	HTM no locks	82	84	66	64	53	65	90	90	92	90	92	90
	SGL fall-back	18	16	34	36	47	35	10	10	8	10	8	10
Baseline	HTM no locks	83	83	75	77	80	79	92	92	94	94	95	97
	SGL fall-back	17	17	25	23	20	21	8	8	6	6	5	3
SCM	HTM no locks	68	65	64	58	68	68	58	60	54	57	51	51
	HTM + Aux lock	18	20	30	35	27	28	36	33	40	36	42	42
	SGL fall-back	14	15	6	7	5	4	6	6	6	7	7	7
Seer	HTM no locks	84	78	84	59	88	76	95	94	96	95	98	96
	HTM + TxLocks	2	5	5	8	5	0	3	3	3	2	1	0
	HTM + CoreLocks	0	2	0	12	0	5	0	0	0	0	0	1
	HTM + Tx + CoreLocks	0	0	0	1	0	3	0	1	0	1	0	1
	SGL fall-back	14	15	11	20	7	16	2	2	2	2	1	2

Table 6.7: Distribution of transaction duration (processor cycles) across benchmarks.

Benchmark	50^{th} perc	90^{th} perc	95^{th} perc	99^{th} perc
Genome	10k	12k	13k	55k
Intruder	$<$ 1k	7k	12k	22k
Kmeans-High	$<$ 1k	$<$ 1k	$<$ 1k	$<$ 1k
Kmeans-Low	$<$ 1k	$<$ 1k	$<$ 1k	$<$ 1k
SSCA2	$<$ 1k	$<$ 1k	$<$ 1k	1k
Vacation-High	14k	22k	25k	42k
Vacation-Low	10k	15k	17k	24k
Yada	$<$ 1k	68k	83k	108k
STMBench7-R	$<$ 1k	130k	132k	133k
STMBench7-RW	$<$ 1k	3k	131k	133k
STMBench7-W	1k	4k	7k	130k

that persists across benchmarks. As for SCM, we can see that it relies very little on the single-global lock, because it ends up replacing it with the auxiliary lock most of the time.

When comparing Baseline with Seer, we can see that the latter reduces a few percentage of the usage of the single-global lock (1% at 14 threads and 4% at 28 threads). This is an interesting result, because we have verified that the performance gains can be quite substantial, e.g., on Genome, despite having only a small effect on the global lock usage. The main reason for this is that not all transaction types are born equal: some generate much longer transactional execution than others.

This phenomenon is quantified in Table 6.7, which reports data on the distribution of the processor cycles spent to execute transactions (serially, without any instrumentation) across all the considered benchmarks. These data highlight that, for Genome, the largest 1% transactions take 5× longer to execute than the median. As a result, even though Seer may prevent only a small portion of transactions to use the single-global lock, the consequence may be a considerable performance improvement in case those transactions are much longer than the median.

Analogously to Genome, we can then see that Intruder and the STMBench7 workloads evidence exactly the same pattern: the single-global lock reduction of Seer is not astounding, but in contrast we can see a large discrepancy between the median and $> 90^{th}$ percentiles (of up to $> 100\times$ difference in transaction length).

With both Vacation workloads we verify that the transaction lengths are more homogeneous.

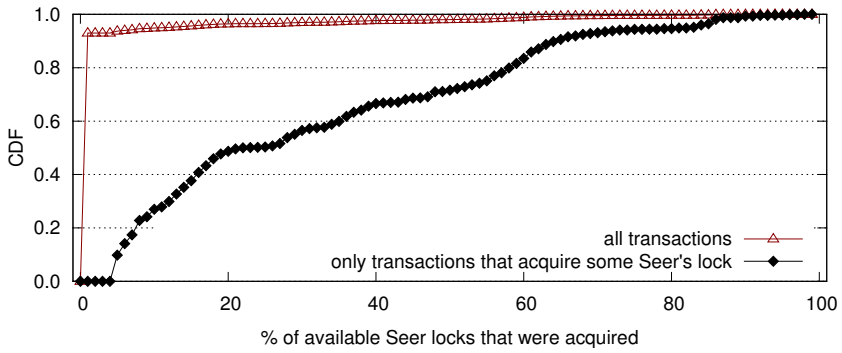


Figure 6.10: Cumulative distribution function for the ratio (in %) of available locks that SEER chose to take, averaged across all the scenarios evaluated in this section. We show this distribution for all transactions, and also for the subset of transactions (on average 7%) that acquires at least one SEER lock.

In fact, in these workloads, the performance gains of SEER are directly related to a significant reduction of the usage of the single-global lock: 14% in Vacation-High and 13% in Vacation-Low, when compared to Baseline at 14 threads (which is close to the peak of speedup).

As for the Kmeans workloads, we had previously seen that SEER did reduce transactional conflicts. This new data now clarifies that such result is achieved thanks to the significant usage of transaction locks (up to 24% in Kmeans-High) and a negligible usage of core locks (given that there are almost no capacity aborts). Yet, as discussed earlier, we now confirm in Table 6.7 that the transactions are very small and largely homogeneous in these workloads. Because of these reasons, the usage of the single-global lock has a minor impact on performance, nullifying the gains of SEER.

We conclude this analysis by presenting, in Table 6.8, average results across all benchmarks, but reporting data for a larger spectrum of thread counts. These results confirm that HLE is the least effective solution among the considered alternatives, with the largest usage of the single-global lock (mainly due the lemming effect [Dice et al., 2008]). Baseline improves over those results, but it still yields an average usage of the single-global lock that is larger than 10% in most cases. SCM reduces that substantially to at most 3%, but at the cost of using more than 10% of the time the auxiliary lock. This is only slightly better than queuing all transactions on the single-global lock.

Finally, SEER is able to improve over all previously described alternatives, exactly because the frequency with which it uses a single-global lock is lower: at most 9% average for 28 threads.

Table 6.8: Breakdown of percentage (%) of types of transactions used on average across all benchmarks.

Variant	Execution Mode	4t	8t	12t	16t	20t	24t	28t
HLE	HTM no locks	80	70	71	75	76	75	75
	SGL fall-back	20	30	29	25	24	25	25
Baseline	HTM no locks	96	94	91	89	88	88	88
	SGL fall-back	4	6	9	11	12	12	12
SCM	HTM no locks	84	82	82	82	80	82	79
	HTM + Aux lock	13	16	16	15	17	16	19
	SGL fall-back	3	2	2	3	3	2	2
Seer	HTM no locks	97	94	91	85	82	80	81
	HTM + Tx Locks	1	2	4	6	7	8	7
	HTM + Core Locks	0	0	0	1	1	2	2
	HTM + Tx + Core Locks	0	0	0	1	2	1	1
	SGL fall-back	2	4	5	7	8	9	9

Furthermore, the other locks that Seer exploits have a much finer granularity — one per transaction and one per core. Hence, unlike the single-global or the auxiliary locks, Seer's locks avoid serializing all transactions.

To complement the results so far, we also show in Figure 6.10 the cumulative distribution function for the ratio of available locks that Seer decides to acquire, averaged across all the tested scenarios. Among all transactions, we note that only about 7% acquires at least one Seer lock, when averaging across benchmarks and number of threads. We highlight that in 50% of these cases, in which *some* lock of Seer is acquired, the fraction of locks that are actually acquired is lower than 23% of the available ones. This experimental result confirms the ability of the proposed lock inference mechanism to synthesize effective fine-grained locking schemes.

6.7.4 What is the Overhead of Running Seer?

We first assess the overhead of the monitoring, lock-inference and self-tuning mechanisms of Seer. For this, we ran a variant of Seer that incurs the overheads of all its mechanisms, without, however acquiring any lock.

In Figure 6.11, we show the average speedup of this Seer's variant relatively to Baseline (that consistently performed second best in our previous evaluation results). In this new plot

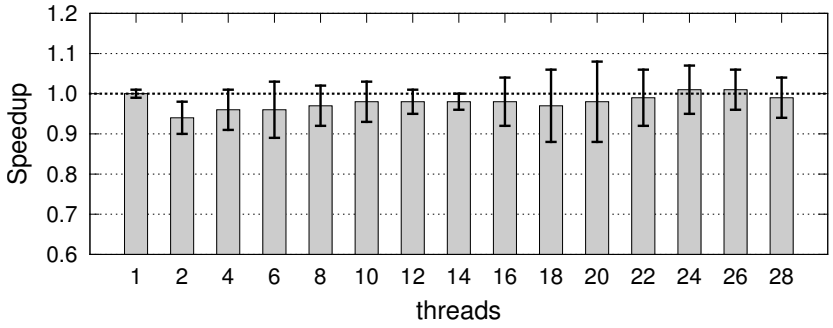


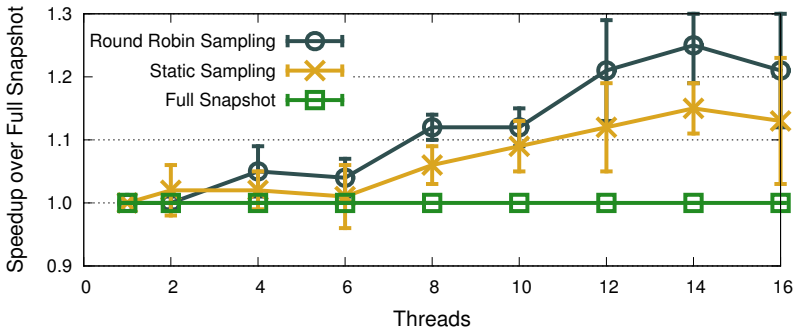
Figure 6.11: Geometric mean overhead of SEER, when profiling and calculating locks to acquire, across **all benchmarks and workloads**.

we present the geometric mean across all benchmarks (of STAMP and STMBench7). The mean slowdown across all these number of threads is of 2%, which confirms that the infrastructure used by SEER to gather data and perform statistical inference is, in fact, quite lightweight. The peak overhead is actually with 2 threads, with 6% slowdown, because in that case we enable the infrastructure of SEER— which is disabled when there is only one thread — and thus half of the threads (i.e., 1 thread) now periodically runs the algorithms for devising the lock scheme and self-tuning the thresholds. As the number of threads increases, this overhead is amortized by the presence of additional threads that execute transactions and merely increase their thread-local statistics counters.

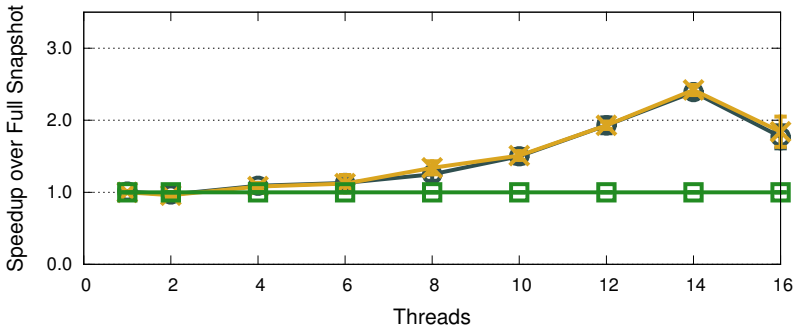
Note that the technique used to sample the *activeTxs* list has the utmost importance in keeping these overheads low as we increase the number of threads. We recall that each thread, upon committing or aborting with a conflict, samples the transaction being executed by only one other thread in the system. This is in contrast with the alternative extreme approach of obtaining all transactions of all other threads (an approach that we call *Full Snapshot*). The problem with this latter alternative is that its overheads increase with the number of concurrently active threads. As such, the intent of sampling just one entry of *activeTxs* is to keep those overheads constant.

In Figure 6.12 we show the speedup of SEER's approach in *Round Robin Sampling* when compared to that of the *Full Snapshot* approach. We also show a slightly different alternative that we call *Static Sampling*, in which each thread always samples one given (different) thread, instead of changing the sampled thread over time as *Round Robin Sampling* does.

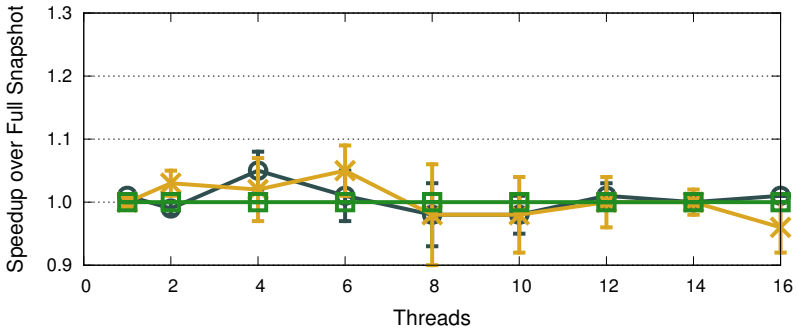
The averaged results (across all benchmarks) shown in Figure 6.12(a) clearly show that *Round Robin* yields improvements over the two alternatives, more so as the number of threads



(a) Average across **all benchmarks and workloads**.



(b) Workload with small transactions and abort ratio (SSCA2 from STAMP).



(c) Workload with large time-dominant transactions (STMBench7).

Figure 6.12: Comparison of three schemes to sample the transactions running concurrently in SEER. In all cases SEER runs the same algorithm, and changes only the sampling scheme of the *activeTxs* list.

increases. The improvements over *Full Snapshot* are explained by the reduced overheads, as shown in Figure 6.12(b) for the benchmark SSCA2, where there are almost no aborts: both sampling alternatives provide considerable speedup over a *Full Snapshot*. Note that this is not necessarily always the case, as shown in Figure 6.12(c), where we use an STMBench7 workload: this has very large transactions, and thus the overhead of sampling at the end of the transaction is amortized significantly in contrast with workloads such that of SSCA2. As a consequence, the three sampling schemes end up having very similar performance.

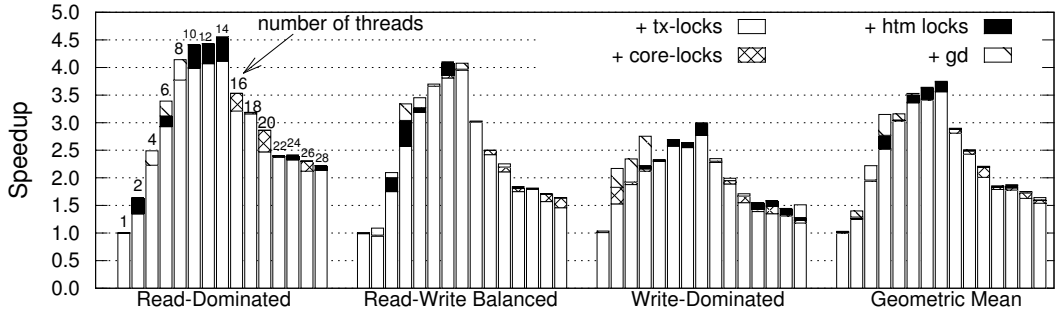
Finally, the difference in performance between both sampling techniques is explained by the better quality of sampling in conflict-prone workloads: by using round-robin over all threads, for instance, this prevents cases where a thread constantly samples the same active concurrent transaction in a thread, which is struggling to progress due to aborts (perhaps due to capacity overflow). As such, *Round Robin Sampling* appears to be the most robust solution, by providing higher quality sampling (over *Static Sampling*), while at the same time reducing the overheads (over *Full Snapshot*).

6.7.5 How Much does Each Design Choice Contribute to Seer?

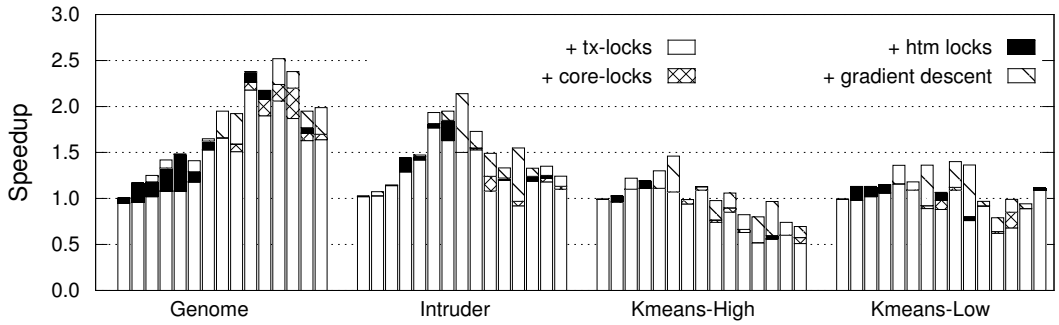
The design of Seer encompasses the following five aspects: 1) capturing statistics about commits, aborts, and concurrent transactions; 2) acquiring transaction locks when aborts occur; 3) acquiring a core lock when a capacity abort happens; 4) acquiring transaction locks by using a hardware transaction to reduce the overheads of multiple compare-and-swaps; and 5) adapting the thresholds $\mathcal{T}h_1$, $\mathcal{T}h_2$ via a stochastic hill climbing algorithm.

To quantify the relative relevance of each of the mechanisms integrated in Seer, we conducted a series of experiments, whose results, shown in Figure 6.13, evaluate the speedup of different variants of Seer. We consider as baseline, the Seer variant previously considered for the plots in Figure 6.11, which incurs the costs of collecting statistics and updating the locking strategy, without ever acquiring any lock. Then, we consider four, progressively enhanced variants, where we cumulatively add: the transaction locks acquisition (*+ tx-locks*), the core locks acquisition (*+ core-locks*), the acquisition of locks by employing a hardware transaction (*+ htm locks*), and the adaptation of the thresholds via hill climbing (*+ hc*).

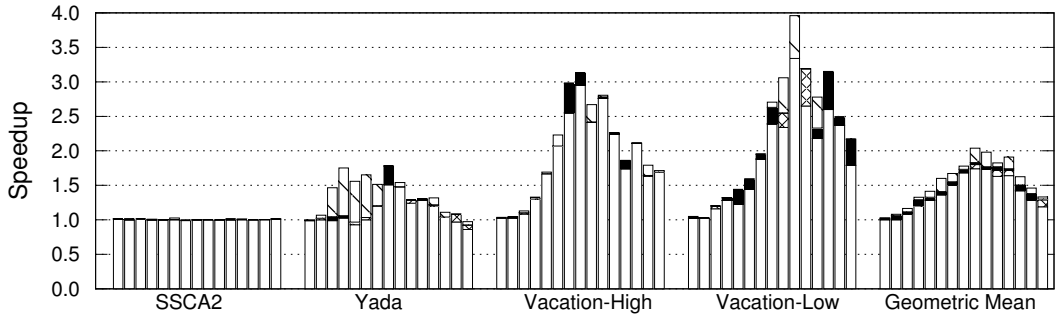
In general the transaction locks provide the largest boost in performance. Unsurprisingly,



(a) STMBench7 workloads.



(b) STAMP benchmarks (1/2).



(c) STAMP benchmarks (2/2).

Figure 6.13: Cumulative contribution of each technique employed in SEER: speedups are shown relatively to SEER without any lock acquisition (but with all the profiling enabled) and the three variants are incrementally added.

the core locks are only beneficial when using 16 or more threads, i.e., when we start executing multiple hardware threads on the same core. Note that, although we strive to reduce both conflict and capacity aborts, the latter are many times unavoidable independently of preventing cache sharing in the same physical core via the core locks. Instead, many of the data conflicts are effectively prevented by constraining the parallelism via the fine-grained transaction locks.

The hardware lock acquisition also shows improvements in general across all numbers of threads. Also, a similar gain is provided by adapting on-line the thresholds used in the transaction locks probabilities calculations.

On average, when considering the geometric mean in the last plot, we get that: 1) transaction locks yield an average improvement of up to 74%; 2) on top of which the core locks add an improvement of up to 10%; 3) the lock acquisition via hardware transactions adds an improvement of up to 9%; and 4) the stochastic hill climbing adds an improvement of up to 21%.

6.8 Summary

In this chapter we proposed Seer, the first *fine-grained* scheduler designed to cope with the specific challenges arising with HTMs, i.e., the lack of detailed knowledge on the root causes of data contention and consequent aborts. The most innovative feature of this proposal is that it can infer conflict patterns among pairs of transactions of a TM program without relying on the availability of precise information from the underlying TM system or static information about the workload to be executed. Seer avoids these pitfalls by relying on lightweight, yet inherently imprecise techniques, in order to gather information on the set of concurrently active transactions upon the commit and abort events of transactions. Probabilistic techniques are then used to filter out false positives and infer a dynamic locking scheme that is used to serialize contention-prone transactions in a fine-grained fashion.

We evaluated our solution, against several alternatives and in various scenarios, using one of the largest HTM-enabled processors available from Intel. In these experiments we verified that Seer yields average improvements of 65% across the various benchmarks and concurrency degrees, with speedups of up to 3.6×. The key reason motivating these performance improvements lies in the reduction of aborts of hardware transactions, which not only decreases the repetition of work, but also diminishes the reliance on the fall-back single-global lock.

7 ProteusTM: an Adaptive High-Performance TM system

In the previous chapters we have focused on improving the performance of either STMs or HTMs in order to address various issues identified in the initial study in Chapter 3. Indeed, the main motivation to consider both types of TM systems in this dissertation is exactly our initial study showed that neither type of TM can outperform the other across different types of workloads. That means we can always find a different workload for an application in which we would prefer to use a different TM algorithm than the one we set out with in the first place.

To some extent, the idea of HyTMs is intended to cover these scenarios, in which the combination of a STM and HTM algorithm would allow to run whichever type of transactions was most adequate to the current workload. However, it is important to highlight that this does not suffice for the issue identified above; i.e., combining two algorithms may still not deliver the optimal performance across workloads, as perhaps combining a different STM with the HTM available in the processor used could work better. Even more importantly, our study in Chapter 3 also identified the sub-optimality of HyTMs due to their inherent overheads stemming from the synchronization needed between the two algorithms, which has been recently studied also from a theoretical point of view [Alistarh et al., 2015].

In this chapter we propose to tackle these problems with a unified TM system that delivers high-performance across workloads by adapting between multiple dimensions of the configurations available. One example dimension is the TM algorithm itself (e.g., choosing between various STM implementations, the HTM available in the processor, or even HyTMs). Other dimensions of the TM configuration are the number of threads allowed to execute concurrently, as well as the retry policy and contention management between transactions.

To accomplish this idea, we propose ProteusTM, which chooses between all configurations the one that is deemed to provide the best performance, via an off-line trained machine learning model. This is achieved with little exploration of configurations (whose number is large due to the exponential nature of the multiple dimensions of the problem) without sacrificing optimality.

7.1 Problem

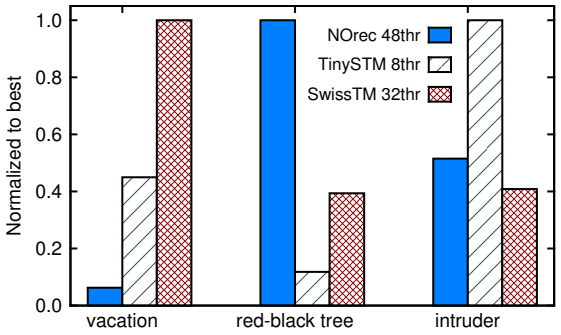
As established by the premise of this dissertation, the abstraction of TM has been widely accepted as a significant improvement to ease the development of concurrent applications with shared memory. However, its performance did not always deliver accordingly to the large expectations, which were amplified by the potential available with optimistic concurrency control that allows a fine-grained synchronization between threads.

So far, we have provided contributions to improve that performance, both for STMs in Chapter 4, as well as for HTMs in Chapter 5 and 6. However, we have also seen in Chapter 3 that neither STMs or HTMs are universally superior to each other. This argument is clearly conveyed in a summarized visualization in Figure 7.1. There, we show the strong sensitivity of different TMs to the workload characteristics. We report on the normalized throughput (in Figure 7.1(a)) and energy efficiency (in Figure 7.1(b)) of various TMs in different machines and benchmarks. We normalized the data with respect to the best performing configuration for the considered workload.

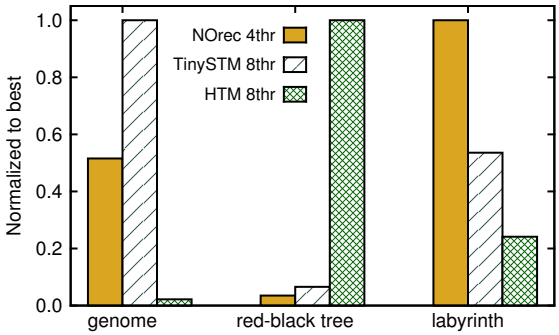
The underlying message of these examples is that the optimal TM choice and configuration differ significantly for each workload. Furthermore, choosing wrong configurations can cripple performance by several orders of magnitude.

The problem is that the efficiency of existing TM implementations is strongly dependent on the workloads they face, meaning that their performance is not very robust to changes. Performance can be affected by a number of factors, including program inputs (i.e., workloads) as demonstrated in Chapter 3, phases of program execution [Didona et al., 2013], tuning of the internal parameters of the TM algorithms as shown in Chapter 5, as well as architectural aspects of the underlying hardware [Castro et al., 2014].

Given the vast TM design space, manually identifying optimal configurations, using trial and error on each workload, is a daunting task. Overall, the complexity associated with tuning TM contradicts the motivation at its basis, i.e., to simplify the life of programmers, and represents a roadblock to the adoption of TM as a mainstream paradigm [Kleen, 2014]. This urges for methodologies capable of automating the identification of the right TM implementation and its proper tuning.



(a) Throughput on a multi-chip 48-core CPU (similar to the machine used in Chapter 4).



(b) Energy efficiency on a single-chip 8-core CPU (similar to the machine used in Chapter 5).

Figure 7.1: Performance heterogeneity in TM applications. We show configurations with different TM algorithms and number of threads used, which results in largely varying degrees of performance, ranging from optimal to considerable slowdowns.

7.2 Overview

We propose a new TM system, ProteusTM[1], which allows developers to still enjoy the simplicity and ease of usage of the TM abstraction, while sparing them from the burden of tuning TM implementations to specific workloads.

Under the simple and elegant interface of TM, ProteusTM hides a large library of TM implementations. At runtime, ProteusTM relies on an innovative combination of learning techniques to pursue optimal efficiency via multi-dimensional adaptation of the TM implementation and its parameters.

At the heart of ProteusTM lie two key components:

• **PolyTM** is a polymorphic TM library that encapsulates several different algorithms from the state of the art of research in TM, and has the ability to transparently and dynamically adapt across multiple dimensions: 1) switch between different TM algorithms; 2) reconfigure the internal parameters of a TM; and 3) adapt the number of threads concurrently executing transactions.

• **RecTM** is in charge of determining the optimal TM configuration for an application across the multiple dimensions mentioned above. Its basic idea is to cast the problem of identifying such best configuration as a recommendation problem [Rajaraman and Ullman, 2011] (similar to the

[1] *Proteus* is a Greek god who can foretell the future and adapt his shape to many forms.

task of a web store that recommends new items based on prior purchases). This allows RecTM to inherit two highly desirable properties of state of the art Recommender System (RS) algorithms: the ability to operate with very sparse training data, and to require only the monitoring of the Key Performance Indicator (KPI) to be optimized. In our case, this KPI can be simply the throughput of the system. This avoids intrusive instrumentation [Rughetti et al., 2012] and (possibly inaccurate) static code analysis [Wang et al., 2012b] employed by other machine learning-based solutions.

While building ProteusTM, we addressed several challenges, with the introduction of a set of innovative solutions:

► *Minimizing the cost of adaptivity.* Supporting reconfigurations across multiple dimensions requires introducing some degree of synchronization, in order to ensure correctness during runtime adaptations. The challenge here is to ensure that the overheads to support adaptivity are kept small enough not to compromise the gains achievable via our self-tuning.

We addressed this challenge by designing lightweight synchronization schemes that exploit compiler-aided, asymmetric code instrumentation. The combination of these techniques allows PolyTM to achieve a negligible overhead of around 1% and a maximum overhead of 5%, even when considering the most performance sensitive TM implementations.

► *Transparency and portability*: PolyTM encapsulates a wide variety of TM implementations, along with their corresponding tuning procedures. The key challenge here is to conceal these mechanisms without breaking the clean and simple abstraction of TM. Furthermore, one of the key design goals of ProteusTM is to seamlessly integrate with existing TM applications, and to support different machine architectures.

We tackled this issue by integrating PolyTM in GCC, via the standard TM ABI [Ni et al., 2008], and by exposing to programmers standard C++ TM constructs. Not only this preserves the simplicity of the TM interface, but it also maximizes portability due to the widespread availability of GCC across architectures.

► *Applying Recommender Systems to the TM domain*: Decades of research have established RS as a powerful tool to perform prediction in various domains (e.g., music and news) [Linden et al., 2003, Davidson et al., 2010, Das et al., 2007]. The application of RS techniques to performance prediction of TM applications, however, raises unique challenges, which were not addressed

by previous RS-based approaches to the optimization of systems' performance [Delimitrou and Kozyrakis, 2014, Delimitrou and Kozyrakis, 2013]. One key issue here is that, in conventional RS domains (e.g., recommendations of movies), users express their preferences on a homogeneous scale (e.g., 0 to 5 stars). On the contrary, the absolute value of KPIs of TM applications can span very heterogeneous scales. As we shall discuss, this can severely hinder the accuracy of existing RS techniques.

We cope with this issue by introducing a novel normalization technique, called *rating distillation*, which maps heterogeneous KPI values to scale-homogeneous ratings. This allows ProteusTM to leverage state-of-the-art RS algorithms even in the presence of TM applications whose KPIs' scales span across different orders of magnitude.

▶ *Large search space*: Although RS algorithms are designed to work with very sparse information, their accuracy can be strongly affected by the choice of the configurations [Su and Khoshgoftaar, 2009] that are initially sampled to characterize a TM application. Deciding *which* and *how many* TM configurations to sample is a challenging task, as ProteusTM supports reconfigurations across multiple dimensions, resulting in a vast search space.

RecTM addresses this issue by relying on Bayesian optimization techniques [Brochu et al., 2010] in order to steer the selection of the configurations included in the characterization of a TM application. This reduces by up to 4× the duration of the learning process of the RS using Collaborative Filtering (CF) [Su and Khoshgoftaar, 2009].

We conducted an extensive evaluation of ProteusTM across all the benchmarks and applications presented in Section 2.7, with a TM parameter space of up to 130 configurations, and optimizing performance in two different machines. Our results highlight that ProteusTM obtains quasi-optimal performance (on average $< 3\%$ from optimal) and gains up to 2 orders of magnitude over static alternatives.

We highlight the fact that ProteusTM was developed in cooperation with another doctoral student, Diego Didona, whom we acknowledge as the main author for the RecTM component, with PolyTM being developed primarily by the student authoring this dissertation. We present the full ProteusTM in this dissertation for self-containment, having only been published before in the conference paper listed under the contributions of this document.

The rest of this chapter is structured as follows. In Section 7.3 we first provide some ad-

ditional background and then discuss related work for optimizing TM systems and for the exploitation of RS for performance prediction. Then, Section 7.4 provides a high-level description of the architecture of ProteusTM, which we detail in Section 7.5 and 7.6. Finally, our evaluation study follows in Section 7.7.

7.3 Background and Related Work

Before specifically comparing our new proposal with the state of the art, we first provide background on the field of Recommender Systems in Section 7.3.1, which is crucial to understanding some of the details of ProteusTM. We then compare our proposal with other systems in the field of optimization for TM (in Section 7.3.2 and with general purpose performance prediction systems that rely on Recommender Systems (in Section 7.3.3).

7.3.1 Collaborative Filtering in Recommender Systems

In general, a Recommender System (RS) seeks to predict the rating that a user would give to an item. These ratings can be exploited to recommend items of interest to users [Linden et al., 2003].

We focus on Collaborative Filtering (CF) [Su and Khoshgoftaar, 2009], a prominent prediction technique used in a RS. To infer the rating of a ⟨user, item⟩ pair, CF techniques exploit the preferences expressed by other users, and ratings by the user on different items. Ratings are stored in a *Utility Matrix* (UM): rows represent users and columns represent items. Typically, a UM is very sparse, as a user rates a small subset of the items. A CF algorithm reconstructs the full UM, from its sparse representation, by filling empty cells with ratings close to the ones that the users would give.

K-Nearest Neighbors (KNN) and Matrix Factorization (MF) are popular techniques to implement the ideas behind CF [Rajaraman and Ullman, 2011]. KNN uses a *similarity function* to express the affinity of two rows or columns: a recommendation for a pair ⟨u,i⟩ is computed with a weighted average of the ratings of the most similar users to u (and/or on the most similar items to i).

MF, instead, maps users and items to a latent factors space of dimensionality d. Each

dimension represents a hidden similarity concept: in the movies' example domain, a similarity concept may be how much a user likes drama movies, or how much a movie belongs to the drama category. To compute recommendations, MF infers two matrices P and Q, which represent, respectively, users and items in the aforementioned d-dimensional space. The product of P and Q is a matrix R that is similar to a given UM A, i.e., $Q^T P = R \sim A$, containing also predictions for the missing ratings in A [Rajaraman and Ullman, 2011].

7.3.2 Optimization of TM Systems

The most researched problem in TM self-tuning is probably that of choosing the number of active threads. Proposed solutions have relied on analytical modeling [Sanzo et al., 2013], off-line machine learning [Rughetti et al., 2012], or exploration-based strategies [Didona et al., 2013]. Another challenging problem is that of choosing the best TM for a given application and workload. In this scope, AutoTM [Wang et al., 2012b] relies on Artificial Neural Networks and programmer heuristics (although they considered only STMs). AutoTM uses a combination of static analysis and runtime profiling, to extract many features of the application, which are then fed to the Neural Networks. We highlight also our previous work with Tuner on Chapter 5, which uses hill climbing and reinforcement learning to adaptively determine the best retry-on-abort policy for HTM.

The main common aspect of these solutions is that they optimize a single characteristic of the TM system. When considering the most challenging task targeted in this chapter, namely to simultaneously optimize multiple configurations of a TM, purely exploration-based solutions (such as Tuner and [Didona et al., 2013]) quickly become impractical, as the number of configurations to explore rapidly grows with the parameters to optimize.

ProteusTM, instead, is effective in optimizing in a large space (we considered 130 configurations over 4 parameters). Thanks to the capability of CF techniques to deal with sparse information, and to the use of Bayesian optimization techniques to steer on-line explorations, ProteusTM is able to operate in even higher dimensional problems by simply including more configurations as extra columns of the UM.

An alternative approach adopted in the literature relies on both code analysis and intrusive software instrumentation, which have the objective of building a workload characterization that

Table 7.1: Comparison of systems and techniques to adapt various internal parameters of TM (the first 5 rows) and of systems that perform adaptation using Recommendation Systems (the following 3). With respect to previous adaptive works in TM, ProteusTM is a more complete solution, as it allows for multi-dimension optimization (i.e., an arbitrary number of tuning parameters, examples of which are listed in the table) without relying on full exploration of all possible configurations. When compared to other systems using Recommender Systems, ProteusTM conducts a guided profiling phase, sampling configurations that maximize the knowledge obtained to better classify the workload.

System	Tuning Parameters	Approach	Inputs	Techniques
[Di Sanzo et al., 2012]	number of active threads	analytical model	read- and write-set sizes, execution times, detailed conflicts	online sampling phase followed by choice given by the model
[Rughetti et al., 2012]	number of active threads	machine learning	read- and write-set sizes, execution times, detailed conflicts	collect traning set and train Neural Network, which is queried at runtime
[Didona et al., 2013]	number of active threads	exploration based	throughput	pure hill-climbing queried at runtime
Tuner	HTM retry policy configuration	exploration based	throughput	reinforcement learning in combination with hill-climbing queried at runtime
[Wang et al., 2012b]	STM algorithm	machine learning and programmer heuristics	read and write-set sizes, throughput conflicts and other features	collect training set and static analysis of code to train Neural Network, which is queried at runtime
[Delimitrou and Kozyrakis, 2013] [Delimitrou and Kozyrakis, 2014] [Pettijohn et al., 2014]	job placement in distributed machines for cloud environments	recommender system coupled with ad-hoc algorithm for placement	instructions per second	collect training data over time; run multiple recommenders in parallel to classify different aspects of incoming jobs by testing a fixed number of randomly sampled configurations
ProteusTM	number active threads TM algorithms, HTM retry policy, others...	recommender system plus Bayesian optimization	Key Performance Indicator (KPI)	collect sparse training set for recommender; queried at runtime to decide which and how many configurations to experiment with

is then fed to different types of machine learning algorithms. The need for relying on software instrumentation to obtain such detailed workload characterization information adds complexity to the code of TM algorithms and overhead to the application. ProteusTM, conversely, relies solely on profiling high-level KPIs (such as the throughput of the application), which incurs minimal overhead and maximizes portability. We stress that ProteusTM's work-flow is fully automated, avoiding the need for programmer heuristics (which is used for instance in AutoTM [Wang et al., 2012b]). We summarize these works, and their comparison with ProteusTM, in Table 7.1.

7.3.3 Performance prediction with Recommender Systems

To the best of our knowledge, Paragon [Delimitrou and Kozyrakis, 2013], Quasar [Delimitrou and Kozyrakis, 2014] and U-CHAMPION [Pettijohn et al., 2014] are the only systems relying on RS for performance prediction, job scheduling and resource provisioning. They characterize an incoming job via *random* sampling of a *fixed* number of configurations and then apply MF-based CF.

These works are also summarized in the second last row of Table 7.1. ProteusTM differs from these works in three key aspects: 1) it relies on a novel rating distillation function that identifies similarity patterns among the performances of heterogeneous applications. One noteworthy finding of our work is that this pre-processing step, not used in previous works, is of paramount importance to achieve high accuracy in the TM domain; 2) ProteusTM leverages model-based techniques to determine which and how many configurations to experiment with during the runtime sampling phase of a new workload, which, as we shall show in the later sections, outperforms random sampling of configurations; and 3) ProteusTM integrates both MF- and KNN-based CF, being able to determine the best one to employ, depending on the training data.

7.4 The Architecture of ProteusTM

In essence, ProteusTM applies Collaborative Filtering (CF) to the problem of identifying the best TM configuration that maximizes a user-defined Key Performance Indicator (KPI): e.g., throughput. ProteusTM aims to maximize the efficiency of TM applications by orchestrating a number of TM algorithms and the dynamic reconfiguration of their parameters. We now overview

the architecture of ProteusTM, depicted in Figure 7.2, which enables its self-tuning capabilities. More details shall be provided in the corresponding sections denoted.

• **PolyTM** (in Section 7.5): consists of a Polymorphic TM library comprising various TM implementations. It allows for switching among TMs and reconfigure several of their internal parameters. It exposes transactional operators via an implementation of the standard TM ABI [Ni et al., 2008] (supported, for instance, by GCC [Intel Corporation, 2009]).

• **RecTM** (in Section 7.6): is responsible for identifying the best configuration for PolyTM depending on the current workload. It is composed, on its turn, by the following sub-modules:

1. **Recommender** (in Section 7.6.1): a RS that acts as a performance predictor and supports different CF algorithms. It receives the KPIs of explored configurations from the Controller, and returns ratings (i.e., predicted KPIs) corresponding to unexplored configurations that may be of interest.

2. **Controller** (in Section 7.6.2): selects the configurations to be used and triggers their adaptation in PolyTM. It queries the Recommender with the KPI values from the Monitor, obtaining estimates for the ratings of unexplored TM configurations.

3. **Monitor** (in Section 7.6.3): this module collects the target KPI to: 1) give feedback to the Controller about the quality of the current configuration, and 2) detect changes in the workload, so as to trigger a new optimization phase in the Controller.

7.5 PolyTM: a Polymorphic TM Library

The PolyTM library encompasses a wide variety of TM implementations (such as many of those that we have used in this work). It interacts with compilers, like GCC, via the standard TM ABI [Intel Corporation, 2009]. Each atomic block, written by the programmer using standard C/C++ constructs [Ni et al., 2008], is compiled into invocations to the various modules of ProteusTM.

For every atomic block, GCC inserts a call to *tm_begin* and *tm_end*, which we direct to PolyTM. Also, two code paths are generated: a non-instrumented path (in terms of reads and writes), and a second one in which reads and writes to memory are instrumented with calls to

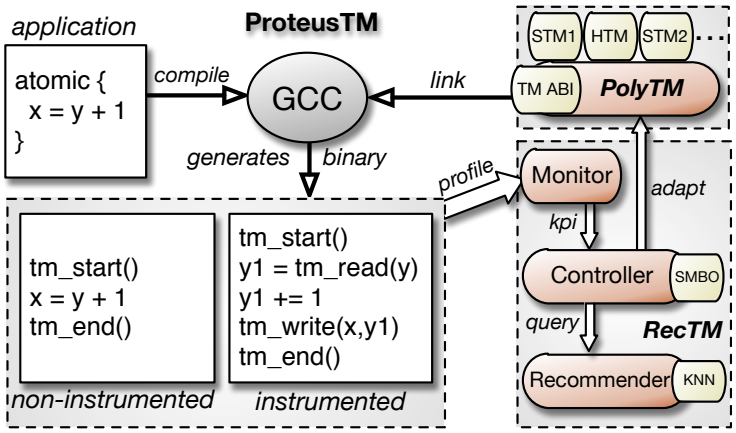


Figure 7.2: Architecture of the ProteusTM system. Applications are written with the canonical atomic blocks used for the TM standard in C/C++. Our system, implementing the standard TM ABI, is linked by the compiler and generates two paths of execution for each atomic block in the application. This leads the application to invoke PolyTM's interface, which hides different TM implementations, and also to support the adaptation of other dimensions. Finally, this adaptation is guided by RecTM, which is fed with the KPI profiling from the lightweight instrumentation placed in the application, and then triggers adaptations in PolyTM.

PolyTM. The latter allows our code to arbitrate reads and writes, besides the begin and commit of transactions, which are common to both paths.

Behind the TM ABI interface, we implemented in PolyTM several TM algorithms, and runtime support to switch among them: 4 STMs [Dalessandro et al., 2010, Dice et al., 2006, Felber et al., 2008, Dragojević et al., 2009a], 2 HybridTMs [Dalessandro et al., 2011, Matveev and Shavit, 2013], and 2 HTMs [Yoo et al., 2013, Adir et al., 2014]. We take advantage of the dual compilation paths and use the instrumented one for the STMs. In contrast, HTMs — which automatically transactionalize reads and writes — execute the non-instrumented one. As shown in Section 7.7.2, the dual path optimization is crucial to minimize overhead.

The compiled code is also instrumented to profile performance metrics in a lightweight and transparent manner. In particular, PolyTM collects the commits and aborts at each thread and the energy consumed by the system, which are examples of possible KPIs. It also uses a dedicated *adapter thread* to change the TM configuration.

In the following, we describe the mechanisms used by PolyTM to support runtime configuration changes. Notably, all of this is performed automatically without the programmer having to worry about anything except writing correct atomic blocks.

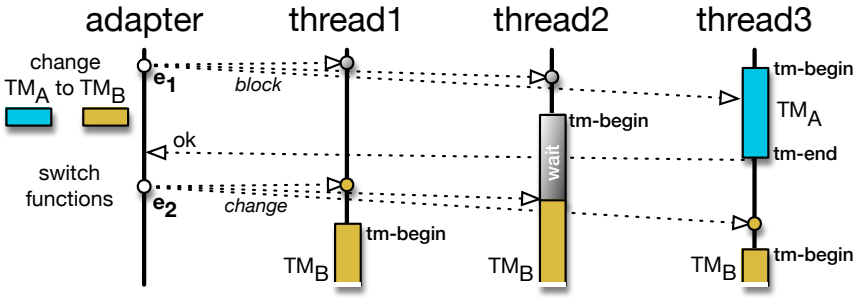


Figure 7.3: Switching TM algorithm safely in PolyTM. In this example we have 3 threads initially executing with TM_A. Then, at event e_1, the adapter thread triggers a change for them to use TM_B. However, performing this change between arbitrary TM algorithms must be done with some care, which is ensured by the adapter thread enforcing some synchronization between the several threads.

7.5.1 Switching Between TM Algorithms

Since our library must interact with the compiler via a single ABI, we hide different TM implementations under a common interface defined in PolyTM. Then, each thread uses a set of function pointers to this interface to process transaction operations. To switch between TMs, a thread switches the function pointers to a different implementation.

Running concurrent transactions with different TMs is not safe in general [Wang et al., 2012b, Lev et al., 2007]. So, PolyTM enforces an invariant: a thread may run a transaction in mode TM_A only if no other thread is executing a transaction in mode TM_B. We illustrate the problem in Figure 7.3: at time e_1 the adapter thread tries to change the TM mode; if thread 2 immediately applied the change, it could run mode TM_B concurrently with thread 1 in TM_A. The above invariant guarantees correctness by forcing thread 2 to wait until e_2 to change to TM_B.

The invariant is enforced via an implementation based on the following three steps: 1) adapt parallelism degree (i.e., number of threads) from its current value, say P, to 0; 2) change TM back-end; 3) adapt parallelism degree back to P.

7.5.2 Adapting the Parallelism Degree

To adapt the maximum number of active threads we use the synchronization scheme described in Algorithm 20.

Each application thread synchronizes with the adapter thread via a (padded) state variable. When executing a transaction for the first time, a thread is registered in PolyTM. We simplify this in the algorithm by assuming a maximum number of threads, although PolyTM supports an arbitrary number.

Upon starting a transaction, a thread t sets the lowest bit in its state variable (line 10), whereas the adapter thread sets the highest bit of t's state variable when it wants to disable t (line 4). These writes are performed atomically together with returning the state of t. Then, both adapter thread and t can reason on who wins (a potential race): if t sees only the lowest bit set, it is allowed to proceed and executes the transaction; otherwise, it must wait for the adapter to change the mode (line 13). The adapter inversely checks that only the highest bit is set, or else waits for t to unset the lowest bit (line 5) — because t was already executing a transaction.

We implement these atomic operations with the primitives fetch-and-add/sub. These primitives always succeed, and are cheaper than the traditional compare-and-swap loop [Morrison and Afek, 2013, David et al., 2013]. Furthermore, in the common case of our algorithm — a thread starting a transaction is not concurrently disabled — each thread performs the atomic operation on a variable residing (with high probability) in its cache and without contention. In this case, the latencies (in processor cycles), in our Machine A are 17 cycles for a fetch-and-add and 32 for a compare-and-swap. As such, the cost for managing the number of active threads is quite limited, for instance when compared to the begin and commit of a hardware transaction (>120 cycles [Ritson and Barnes, 2013]).

We also use a conditional variable, associated with each thread t, for t to wait on, in the case it is disabled. We omit the details of its management, for simplicity of presentation.

PolyTM guarantees that a reconfiguration always terminates: a thread eventually commits a pending transaction, or else aborts and checks whether it was disabled — assuming finite atomic blocks. Hence, the duration of a reconfiguration depends on the longest running transaction. This, however, does not impair the efficiency of PolyTM's reconfiguration: in-memory transactions are generally very fast (given that they do not entail I/O) [Tu et al., 2013, Larson et al.,

Algorithm 20: Changing the parallelism degree in PolyTM.

```
1:  const int RUN ← 1 , BLOCK ← 1 ≪ 32
2:  padded var int threadState[MAX_THREADS] ← { 0 }

3:  function disable-thread(int t)                          ▷ adapter thread
4:    int val ← fetch-and-add(threadState[t], BLOCK)
5:    while (val & RUN) val ← threadState[t]

6:  function enable-thread(int t)                           ▷ adapter thread
7:    threadState[t] ← RUN
8:    signal(t)                          ▷ wakes up thread t (locking omitted)

9:  function tm-start(int t)                                ▷ application thread
10:   int val ← fetch-and-add(threadState[t], RUN)
11:   if (val & BLOCK)
12:     fetch-and-sub(threadState[t], RUN)
13:     cond-wait(t)                     ▷ checks it is still blocked after locking
14:   ▷ ...omitting logic for tm-start...

15: function tm-end(int t)                                  ▷ application thread
16:   ▷ ...omitting logic for tm-end...
17:   fetch-and-sub(threadState[t], RUN)
```

2011].

In addition, the success of a reconfiguration does not rely on threads to eventually call into ProteusTM. This is crucial to cope with applications whose threads may wait for events (e.g., client requests) and do not run atomic blocks often.

We note that, depending on the application, it may not be safe for PolyTM to permanently disable an arbitrary thread: for instance, a web server may have a single thread accepting requests. To account for such cases, in which it is impossible to know the application's semantics, we provide a library call for the programmer to forbid PolyTM from disabling a specific thread (e.g., to tune the parallelism degree). Such a thread, however, may be disabled temporarily to allow switching the TM algorithm, which is a brief procedure as noted above. Depending on the configuration of ProteusTM, this behaviour could be made the default, and specific threads could be identified as possible for adaptation.

7.5.3 Adapting the Contention Management

PolyTM's optimization encompasses other configuration parameters related to contention management [Guerraoui et al., 2005b]. Specifically, PolyTM integrates the configurations that

we presented initially in the scope of Tuner for HTM in Chapter 5. Those configurations are affected by two parameters: 1) the budget of retries using HTM for a transaction, and 2) whether, upon a capacity abort, the budget should be decreased by one, halved, or fully consumed.

In fact, different contention management policies can co-exist without threatening correctness [Guerraoui et al., 2005b]. Hence, both parameters can be changed at any point without synchronization.

This allows ProteusTM to optimize also these important characteristics of HTM in the case of deploying for multiple applications or varying workloads that cannot be optimized solely with Tuner, namely because they require a varying number of active threads or different TM algorithms at different phases of the workload.

7.6 RecTM: a Recommender System for TM

RecTM is the component responsible for driving the decisions that trigger the adaptations in PolyTM. The methodology employed uses a *black-box* learning approach that relies on a novel combination of off-line and on-line learning. In short, it operates according to the work-flow of Algorithm 21:

1. Build a *training set* by profiling the KPI of an initial set of applications in the encompassed TM configurations (line 1).

2. Instantiate a CF-based performance predictor based on the training set obtained off-line in 1) (lines 2 and 3).

3. Upon deploying a new application or detecting a change of the workload, profile on-line the application over a small set of explored configurations (lines 4 and 5).

4. Recommend a configuration for the workload (line 6).

In the following sections we provide details about the building blocks of RecTM by describing their functionalities and interactions.

Algorithm 21: RecTM work-flow to drive the adaptation of ProteusTM.

1: Off-line performance profiling of an initial training set of applications.
2: Rating distillation and construction of the Utility Matrix (Section 7.6.1).
3: Selection of CF algorithm and setting of its hyper-parameters (Section 7.6.1).
4: Upon the arrival of a new workload (Section 7.6.3):
5: Sample the workload on a small set of initial configurations (Section 7.6.2).
6: Recommend the optimal configuration (Section 7.6.1).

7.6.1 Recommender: Using Collaborative Filtering

RecTM casts the identification of the optimal TM configuration for a workload into a recommendation problem, which it tackles using Collaborative Filtering (CF), an efficient and simple technique for rating prediction [Su and Khoshgoftaar, 2009].

A key challenge to successfully apply CF in predicting the performance of TM applications, is that CF assumes the ratings in a predetermined scale (e.g., a preference from 0 to 10). However, the absolute KPI values produced by different TM applications, instead, can span orders of magnitude (e.g., from millions [Minh et al., 2008] to few transactions per second [Guerraoui et al., 2007]). Further, KPI values of specific configurations provide no indication on the maximum/minimum KPI that the application can obtain, impairing their normalization.

Our Recommender tackles this issue with an innovative technique, which we call *rating distillation*. This function maps KPI values of diverse TM applications onto a rating scale that can be fruitfully exploited by CF to identify correlations among the performance trends of heterogeneous applications. In the following we discuss the difficulties associated with obtaining this function and its use in Recommender.

The Rating Heterogeneity Problem. Ratings are stored in a Utility Matrix (UM) A, of which each row u represents a workload and each column i is a TM configuration: $A_{u,i}$ is the rating of configuration i for workload u (i.e., in our domain, it expresses the performance of i in u for a given KPI metric). To illustrate the problem, we populated an example UM directly with sampled KPI values (e.g., throughput) in Table 7.2. That UM contains information on applications A_1 and A_2 profiled with configurations C_1, C_2 and C_3 and A_3 profiled only at C_1 and C_2. Each configuration varies the number of active threads. From the matrix, we can infer that A_1 can scale, as its performance increases linearly with the number of threads; A_2 does not, since its performance, though higher in absolute value than A_1's, decreases as the number of threads grows. We want to predict the rating for $A_{3,3}$. Note that A_3 exhibits the same linear

Table 7.2: Example of a Utility Matrix where rows represent different applications (or workloads) and the columns represent different configurations of the TM system. In this example we simplify the configurations to vary only the number of active threads being used.

	C_1	C_2	C_3
A_1	1	2	3
A_2	30	20	10
A_3	100	200	**?**

trends of A_1: for this reason, a likely value for $A_{3,3}$ would be 300. Next, we show why well-known CF techniques can be misled because of the heterogeneity of the ratings' scales in the UM.

The Need for Normalization. The most common similarity functions in KNN CF are the Euclidean, Cosine and Pearson [Rajaraman and Ullman, 2011]. The first cannot be applied to heterogeneous ratings, because it is based on the scale-sensitive Euclidean distance: in the example above, it would incorrectly regard C_2 as more similar to C_3 than C_1. The other two are scale-insensitive, so they are able to identify C_1 as similar to C_3. However, they would yield an incorrect prediction in absolute value, as it will lie on C_1's scale, which is different from C_3's.

A similar shortcoming applies to MF CF. The P and Q matrices — recall Section 7.3.1 — are typically obtained by means of stochastic gradient descent [Rajaraman and Ullman, 2011]: starting from random matrices, this technique iteratively tries to minimize the fitting error of P^TQ over A. Thus, it is prone to over-fitting around the highest absolute value ratings, yielding poor overall accuracy.

A solution to these problems is to normalize the entries in UM. An effective normalization function should fulfil two requirements: 1) to transform entries in the UM so that similarities among heterogeneous applications can be mined, and 2) to enable the application of conventional CF techniques.

Note that feature normalization is often performed in Machine Learning (ML): a notable example is that of Artificial Neural Networks, which normalize input features in the range [0,1] [Bishop, 2006]. In ML, however, normalization is performed on the input features, whose values are fully known for samples in the training set and for queries. In contrast, in ProteusTM, the normalization has to be performed on the UM, which contains values corresponding to the output feature KPI, and whose entries are not all known. Next, we describe how ProteusTM nor-

malizes ratings to meet the two aforementioned requirements and, thus, enables CF to optimize TM applications.

Normalization in the Recommender. If the minimum and maximum KPIs of an application were known *a priori*, they could be mapped to a homogeneous rating scale with a simple, per workload, normalization. Since KPIs of applications can take arbitrary values, then this *ideal* solution cannot be used.

The rating distillation used by the Recommender approximates the ideal approach with a mapping function that, for any workload w in the UM, ensures:

1. The ratio between the performance of two configurations c_i, c_j is preserved in the rating space, i.e., $\frac{KPI_{w,c_i}}{KPI_{w,c_j}} = \frac{r_{w,c_i}}{r_{w,c_j}}$.
2. The ratings of the corresponding configurations, $r_{w,c}$, are distributed in the range of [0, max$\{M_w\}$] — assuming a maximization problem — so as to minimize that index of dispersion of M_w, $D(M_w)$= $\frac{var(M_w)}{mean(M_w)}$ (where M_w is defined below).

Property 1) ensures that the information about the relative distances of two configurations is correctly encoded in the rating spaces. Property 2) aligns the scales that express the ratings of each workload w to use similar upper bounds M_w, which are tightly distributed around their mean value.

We define this function in Algorithm 22. The rating of each row is obtained by normalizing its KPI with respect to a column C*$\in \{C_1 ... C_K\}$ (assuming there are K configurations), so to minimize the index of dispersion among the resulting maximum ratings in the normalized domain.

Note that, not only does this function reduce the numerical heterogeneity of ratings, but it also projects all the elements of the matrix to a semantically common domain: now, a rating k for configuration i can be seen as "configuration i delivers performance that are k times the

Algorithm 22: Rating Distillation function in ProteusTM.

```
1: for C_i ∈ C_1 ... C_M
2:     Normalize Matrix KPI w.r.t. C_i
3:     Collect the vector M_w with the max values per row
4:     Compute mean_i(M_w) and var_i(M_w)
5: return C* = argmin_{i∈1...M} var_i(M_w)/mean_i(M_w)
```

reference one". While an absolute throughput of 5000 transactions per second may correspond to either a good or a bad performance depending on the application, our rating function gives ratings a "more universal" meaning. Also, minimizing the dispersion of the maximum values of the scales, allows for aligning the upper extreme of the rating distributions of each application (i.e., a row of the UM) to a common value: the tighter the distribution around a common value M_w, the closer it approximates an ideal "omniscient" normalization.

Tuning the Recommender. We used Mahout [Owen et al., 2011], a ML framework containing several CF algorithms. This design choice allows the Recommender to seamlessly leverage a vast library of CF techniques, rather than binding it to a single one.

The Recommender uses the training UM to choose one of the available CF algorithms, to adopt at runtime, and properly tunes its parameters (e.g., similarity function). Determining the best learning algorithm and its hyper-parameters, given a training set, is a challenge that falls beyond the domain of CF [Bergstra et al., 2011], and the literature abounds of heuristics for automating the search of optimal models' parameterizations. In our Recommender, we use an approach based on random-search [Bergstra and Bengio, 2012] and n-fold cross-validation [Bishop, 2006, Thornton et al., 2013, Hutter et al., 2011].

7.6.2 Controller: Explorations Driven by Bayesian Models

The Controller uses Sequential Model-based Bayesian Optimization (SMBO) [Hutter et al., 2011] to drive the on-line profiling of incoming workloads, to quickly identify optimal TM configurations.

SMBO is a strategy for optimizing an unknown function $f : D \to \mathbb{R}$, whose estimation can only be obtained through (possibly noisy) observation of sampled values. It operates as follows:

1. Evaluate the target function f at n initial points $x_1 \dots x_n$ and create a training set S with the resulting $\langle x_i, f(x_i)\rangle$ pairs.
2. Fit a probabilistic model M over S.
3. Use an *acquisition function* $a(M, S) \to D$ to determine the next point x_m.
4. Evaluate the function at x_m and accordingly update M.

5. Repeat steps 2) to 4) until a stopping criterion is satisfied.

Acquisition function. Our Controller uses as acquisition function the criterion of *Expected Improvement (EI)* [Jones et al., 1998], which selects the next point to sample based on the gain that is expected with respect to the currently known optimal configuration.

More formally, considering without loss of generality a minimization problem, let D_e be the set of evaluation points collected so far, D_u the set of possible points to evaluate in D and $x_{min} = \arg\min_{x \in D_u} f(x)$. Then the positive improvement function I over $f(x_{min})$ associated with sampling a point x is $I_{x_{min}}(x) = max\{f(x_{min}) - f(x), 0\}$. Since f has not been evaluated on x, $I(x)$ is not known *a priori*; however, thanks to the predictive model M fitted over past observations, it is possible to obtain the expected value for the positive improvement:

$$\begin{aligned} EI_{y(x_{min})}(x) &= \mathbb{E}[I_{y(x_{min})}(x)] \\ &= \int_{-\infty}^{\infty} max\{f_{x_{min}} - c, 0\} p_M(c|x) dc \\ &= \int_{-\infty}^{y(x_{min})} (f_{x_{min}} - c) p_M(c|x) dc \end{aligned}$$

Here, $p_M(c|x)$ is the probability density function that the model M associates to possible outcomes of the evaluation of f at point x [Jones et al., 1998].

High EI values are associated either with points that are regarded by the model as likely to be the minimum (high predicted mean), or with points whose corresponding value of the target function the model is uncertain about (high predicted variance).

By selecting as next point for evaluation the one that maximizes the EI, SMBO naturally balances exploitation and exploration: on one side it exploits the model's confidence to sample the function at points that are supposedly good candidates to be the minimum; on the other, it explores zones of the search space for which the model is uncertain, to increase its predictive power by iteratively narrowing uncertainty zones.

Computing $\mathbf{p_M(c|x)}$. The Controller computes $p_M(c|x)$ with an ensemble of CF predictors, and obtains predictive mean μ_x and variance σ_x^2 of $p(c|x)$ as frequentist estimates over the output of its individual predictors evaluated at x.

It then models $p_M(c|x)$ as a Gaussian distribution $\sim N(\mu_x, \sigma_x^2)$. Assuming a Normal dis-

tribution for $p(c|x)$ is frequently done in SMBO [Hutter et al., 2011] and other optimization techniques [Osogami and Kato, 2007] to ensure tractability. Given a Gaussian distribution for $p_M(c|x)$, $EI_{y(x_{min})}(x)$ can be computed in closed form as $EI_{y(x_{min})}(x) = \sigma_x[u\Phi(u) + \phi(u)]$, where $u = \frac{y(x_{min})-\mu_x}{\sigma_x}$ and Φ and ϕ represent, respectively, the probability density function and cumulative distribution function of a standard Normal distribution [Jones et al., 1998].

More in detail, the Controller builds a *bagging* ensemble [Breiman, 1996] of k CF learners, each trained on a random subset of the training set. Then, it computes μ_x as the average of the values output by the single predictors, and σ_x^2 as their variance. In ProteusTM, we use 10 bagged models; we highlight that the cost of employing them instead of a single one is negligible, mainly because they are only queried during profiling phases.

Stopping Criterion. As discussed, SMBO requires the definition of a predicate to stop exploring new configurations. Devising a stopping criterion for the on-line profiling phase in ProteusTM is not an easy task: since the target, optimal KPI value is not known beforehand, ProteusTM cannot exactly determine how distant the best KPI value sampled so far is from the global optimum. Therefore, the stopping predicate must identify a good trade-off between exploitation, i.e., trust in the model, and exploration of additional configurations.

Our Controller uses a stopping criterion that seeks a balance between exploration and exploitation by relying on the notion of EI: it uses the estimated likelihood that additional explorations may lead to better configurations. More precisely, the exploration is terminated after k steps when:

1. The EI decreased in the last 2 iterations $k-2$ and $k-1$.
2. The EI for the k-th exploration step was marginal, i.e., lower than ϵ with respect to the current best sampled KPI.
3. The relative performance improvement achieved in the $k-1$-th iteration (i.e., by exploring the configuration recommended at iteration $k-1$) did not exceed ϵ.

In Section 7.7.3, we evaluate the effectiveness of this policy, comparing it also with a solution that blindly relies on the model's output, and evaluating its sensitivity with respect to the ϵ parameter.

7.6.3 Monitor: Lightweight Behavior Change Detection

The Monitor periodically gathers KPIs from PolyTM. These are used for two tasks:

1. While profiling a new workload, they are fed to the Controller, providing feedback about the quality of the current configuration.

2. At steady-state, they are used to detect a workload change.

The Monitor implements the Adaptive CUSUM algorithm to detect, in a lightweight and robust way, deviations of the current KPI from the mean value observed in recent time windows [Basseville and Nikiforov, 1993]. This allows the Monitor to detect both abrupt and smooth changes and to promptly trigger a new profiling phase in our Controller.

Note that environmental changes (e.g., inter-process contention or VM migration) are indistinguishable from workload changes from the perspective of our behavior change detection.

7.7 Evaluation

This section provides an extensive validation of our contributions. We introduce, in Section 7.7.1 the test-bed, applications, and accuracy metrics used. In Section7.7.2 we assess the overhead incurred by PolyTM to provide self-tuning capabilities. In Section 7.7.3, we evaluate the effectiveness of RecTM's components separately. Finally, in Section 7.7.4 we evaluate the ability of ProteusTM to perform online optimization of dynamic workloads.

7.7.1 Experimental Test-Bed

We deployed ProteusTM in two machines with different characteristics: Machine A, corresponding to that already described in Table 3.2 with HTM support; and Machine B, equivalent to that already described in Table 4.4 but with 48 cores.

We use a wide variety of TM benchmarks and applications encompassing all those presented in Section 2.7: the STAMP suite, STMBench7, concurrent data-structures, TPC-C and Memcached. We considered over 300 workloads for all those benchmarks — varying possible values

Table 7.3: Parameters tuned by ProteusTM in the two different machines. Note that Machine B does not have Intel RTM support.

Machine	TMs Available	# threads	HTM Retry Budget	HTM Capacity Abort Policy
A	TL2, NOrec, TinySTM, SwissTM, RTM-SGL	1,2,3,4, 5,6,7,8	1,2,4, 8,16,20	Set budget to 0; decrease budget by 1; halve budget
B	TL2, NOrec, TinySTM, SwissTM	1,2,4,6, 8,16,32,48	N/A	N/A

for their standard input parameters — which are representative of heterogeneous applications, from highly to poorly scalable, from HTM to STM friendly.

Our system optimizes a given KPI — we generally focus on throughput in this evaluation — by tuning the four dimensions listed in Table 7.3. Note that ProteusTM also includes HyTMs as available TMs. However, we omit them as they were shown in Chapter 3 to perform sub-optimally in comparison to the alternative of running a pure HTM or STM alone. We also created TM support for IBM Power8 HTM, however we omitted it from the results to narrow down the available machines and span of data. Overall,the optimization problem for ProteusTM has a total of 130 TM configurations for Machine A and 32 for Machine B.

Evaluation metrics. We evaluate the accuracy of the optimization conducted by ProteusTM along 2 accuracy metrics: Mean Average Percentage Error (MAPE) and Mean Distance From Optimum (MDFO).

Noting $r_{u,i}$ the real value of the target KPI for workload u, when running with i as configuration, $\widehat{r}_{u,i}$ as the corresponding prediction of the Recommender, and S the set of testing $\langle u, i\rangle$ pairs, then **MAPE** is defined as follows: $\sum_{\langle u,i\rangle \in S} |r_{u,i} - \widehat{r}_{u,i}|/r_{u,i}$.

Noting with i^*_u the optimal configuration for workload u, and with $\widehat{i^*_u}$ the best configuration identified by the Recommender, the **MDFO** for u is computed as: $\sum_{\langle u,\cdot\rangle \in S} |r_{u,i^*_u} - r_{u,\widehat{i^*_u}}|/r_{u,i^*_u}$.

MAPE reflects how well the CF learner predicts performance for an application with respect to their real KPI values. In contrast, MDFO captures the quality of final recommendations output by the Recommender, by comparing the choices taken with the ones known to be optimal.

Table 7.4: Average overhead (%) incurred by ProteusTM for different TM algorithms and number of active threads on Machine A.

#threads	TL2	NOrec	SwissTM	TinySTM	HTM-opt	HTM-naive
1	3	3	2	3	3	14
4	< 1	1	< 1	3	3	14
8	< 1	< 1	< 1	4	5	24

7.7.2 Overhead Analysis and Reconfiguration Latency

We now assess the average overhead of PolyTM (over 10 runs in each case), i.e., the inherent steady-state cost of supporting adaptation. We compare the performance of a bare TM implementation T with that achieved by PolyTM using T without triggering adaptation.

Table 7.4 summarizes the results averaged across all the benchmarks. The retry policy for HTM is set to decrease linearly the retries starting from 5 (a common setting [Yoo et al., 2013,Karnagel et al., 2014]). We also show the overhead of the optimized code path, employed for HTM, and the one resulting from the default GCC instrumentation (fully instrumented path).

These experiments reveal overheads consistently $< 5\%$ across the TM algorithms. There is a slightly lower STM overhead, which is justifiable considering that STMs natively suffer from instrumentation costs that end up amortizing most of the additional overhead introduced by PolyTM. The last column of Table 7.4 clearly highlights the relevance of the dual path optimization, which allows PolyTM to halve the overhead incurred when using HTM.

We also assess the average latency of a typical reconfiguration in PolyTM to switch TM algorithms (which also entails changing the number of threads). The results, shown in Table 7.5, encompass two heterogeneous workloads: Memcached uses 100× shorter transactions than TPC-

Table 7.5: Examples of the latency (in microseconds) to conduct a reconfiguration of a TM algorithm.

Benchmark (Machine)	# Threads					
	1	2	4	8	16	32
TPC-C (Machine A)	21	91	213	3419	N/A	N/A
Memcached (Machine B)	2	8	28	145	1103	1849

C. The results highlight the practicality of our reconfiguration algorithm. Even in the worst case of large transactions in TPC-C, the latency is negligible because such costs are only taken during the exploration phase for new configurations, which, as we shall see, is kept very short by ProteusTM.

7.7.3 Quality of the Prediction and Learning Processes

We now evaluate each component of RecTM by means of a trace-driven simulation. We collected traces of *real* executions of a subset of the test cases (namely, STAMP and concurrent data-structures), averaging the results over 5 runs.

The data-set was split into a training set (30%) and a test set (70%). The training set is used to choose and tune the CF algorithm (recall Section 7.6.1) and to instantiate the predictive model. We used 10 learners for the bagging ensemble, as this is a typical value [Hutter et al., 2011, Thornton et al., 2013]. To simulate sampling the performance of the application in a given configuration, the corresponding value from the test set is inserted in the UM of the Recommender.

The simulation proceeds in rounds, where each round optimizes a workload. Each round starts with the profiling of the target workload on the reference configuration chosen for normalization; then the sampling phase begins. Such phase ends when the EI-based termination predicate described in Section 7.6.2 is evaluated to true. At this point, ProteusTM is asked to produce a recommendation about the optimal configuration for the workload. If such configuration has already been explored, then the optimization of the workload is concluded and the *final* distance from optimum coincides with the distance from optimum corresponding to the best sampled configuration. Otherwise, a final exploration of such suggested configuration c_f is performed: if the sampled KPI with c_f is better than the optimal observed so far c_o, the final distance from optimum is computed on the basis of c_f; if this last exploration turns out to be fruitless, the final distance from optimum is computed on the basis of c_o.

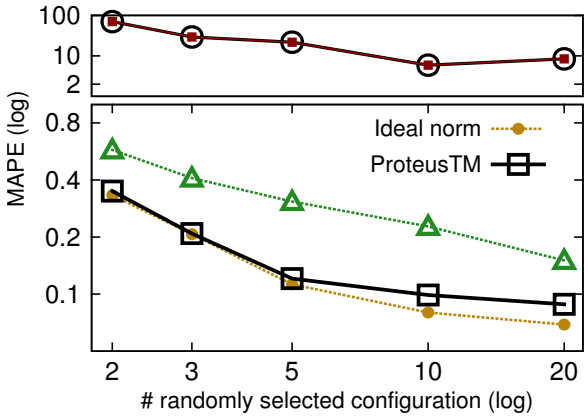
The *final* distance from optimum, as opposed to the plain distance from optimum introduced before, serves the purpose of simulating that ProteusTM would bring the application to work with the best observed configuration, even if it differs by the one recommended after the last exploration step.

Rating distillation. We start by assessing the effectiveness of the UM preprocessing technique employed in ProteusTM. Specifically, we compare ProteusTM's novel data normalization with the following data preprocessing techniques:

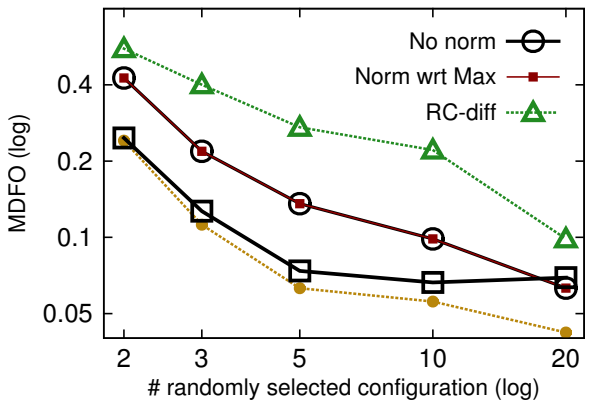
- *No normalization:* CF is applied on the UM containing raw KPI samples. This is equivalent to Quasar [Delimitrou and Kozyrakis, 2014] (see Section 7.3.3).
- *Normalization with respect to the maximum:* entries in the UM are relative to the highest value, assumed to be known a priori. It resembles Paragon's approach [Delimitrou and Kozyrakis, 2013] (see Section 7.3.3), where the machine's peak instructions per second rate is used as normalizing constant; and for which a maximum can be calculated. Such KPI, however, is meaningless in TM applications given the possibility of aborts and transaction restarts.
- *Ideal normalization:* the scheme described in Section 7.6.1, according to which the values in each row are normalized with respect to a value known to be optimal due to an off-line exhaustive exploration of all possible configurations for the workload in particular.
- *Row-column subtraction:* noted *RC-diff* in the plots, is typically employed in CF to cope with biases in users and item ratings [Rajaraman and Ullman, 2011]. It consists in removing from each known rating the average value of the corresponding row. Then, the average value per column — computed after the first subtraction — is subtracted.
- *Rating distillation:* used in ProteusTM (as specified in Algorithm 22).

We present results focusing on the throughput KPI on Machine A and employing KNN with cosine similarity. We vary the number of randomly chosen known ratings per row and compute MAPE and MDFO.

Figure 7.4 shows that using no normalization, or normalization w.r.t. the maximum performs very poorly, both in terms of MAPE (Figure 7.4(a)) and MDFO (Figure 7.4(b)). That is because they are both performing a normalization with respect to some constant that has no meaning in the scope of the applications used. RC achieves lower MAPE than the two aforementioned normalizations, yet its accuracy is significantly worse than that of rating distillation, both in terms of MAPE and MDFO. Also, the approach of ProteusTM closely follows the ideal normalization.



(a) Mean Average Percentage Error (MAPE).



(b) Mean Distance From Optimum (MDFO).

Figure 7.4: Accuracy assessment for the proposed rating distillation function for an average of executions on Machine A with the RecTM instantiated with KNN CF relying on a Cosine similarity function. For both metrics lower values are better.

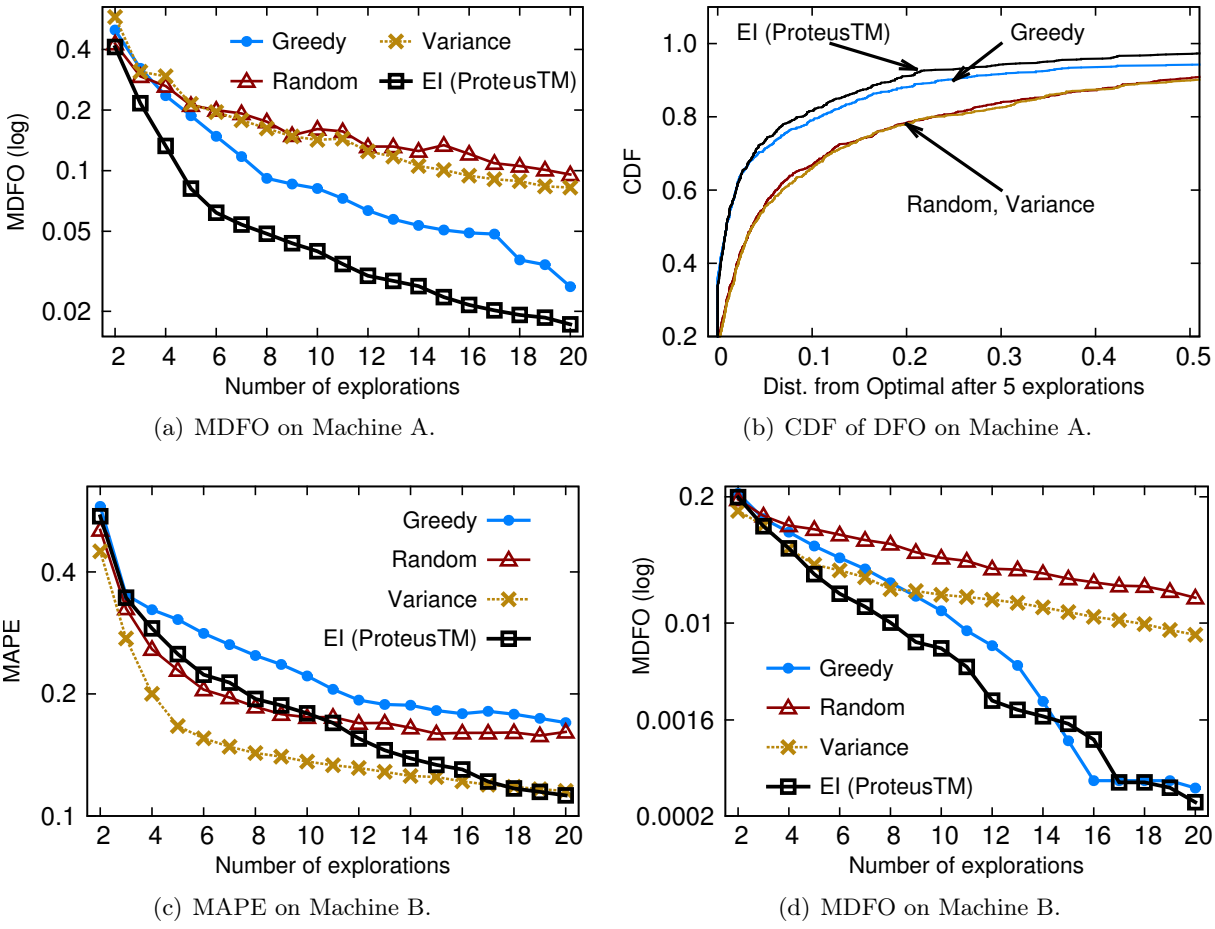
To ensure a fair comparison, we used the same training set, without forcing the presence of the column used for normalization among the profiled configurations for ProteusTM.

We have obtained other similar results with other distance functions in KNN and MF (which is used by other proposals that rely on a RS for performance prediction, see Section 7.3.3). Our results confirm the key role of rating distillation to enable the use of CF in the domain of performance prediction for TM applications.

Controller. We evaluate the effectiveness of our SMBO approach to the sampling of new workloads. We compare our solution (called EI, from Expected Improvement) with a randomized sampling approach, used in Quasar and Paragon [Delimitrou and Kozyrakis, 2013, Delimitrou and Kozyrakis, 2014] (see Section 7.3.3), and two other SMBO approaches using acquisition functions different from ours: Variance explores configurations with high uncertainty for the underlying model (i.e., high *variance/mean* ratio); and Greedy explores the configuration with highest predictive mean.

In Figure 7.5(a), we report the MDFO for the tests executed on Machine A. Our EI exploration policy is able to identify a high quality solution requiring, on average, less explorations than any competitor. Figure 7.5(b) shows the 80^{th} percentile of the DFO obtained by EI — after 5 explorations — is less than 10%.

In Figure 7.5(d), we show the MDFO for Machine B: once again, our EI-based Controller's



(a) MDFO on Machine A.

(b) CDF of DFO on Machine A.

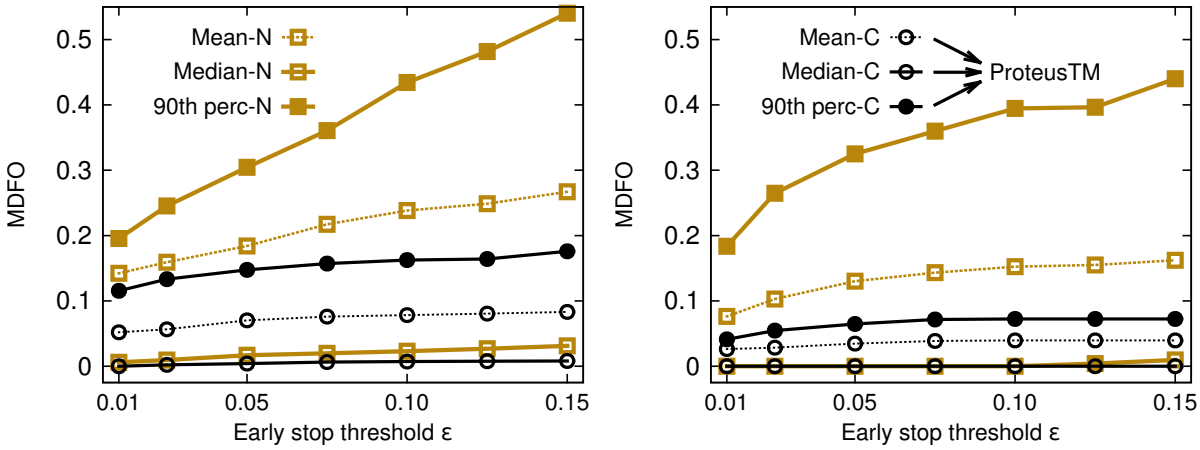
(c) MAPE on Machine B.

(d) MDFO on Machine B.

Figure 7.5: Assessment of the accuracy for the exploration policy used by our Controller in comparison with several alternatives in both machines used in our evaluation study.

exploration performs best. Figure 7.5(c) shows the MAPE per explorations. Interestingly, the Variance policy has the best *mean* prediction accuracy. However, as it does not aim at sampling potential optimal solutions, but only at reducing uncertainty, it does not learn the behavior of the target function for potentially good configurations. Thus, the quality of the recommended configurations is significantly worse than EI's (notice its higher MDFO in Figure 7.5(d)).

Finally, we compare our EI policy with random sampling in Figures 7.5(a) and 7.5(d): taking 5% distance as reference, EI achieves a number of explorations vs MDFO trade-off that is up to 4× better than this random competitor. This highlights the effectiveness of our SMBO-based approach over simpler sampling techniques used in recent systems [Delimitrou and Kozyrakis, 2014, Delimitrou and Kozyrakis, 2013].



(a) MDFO on Machine A.

(b) MDFO on Machine B.

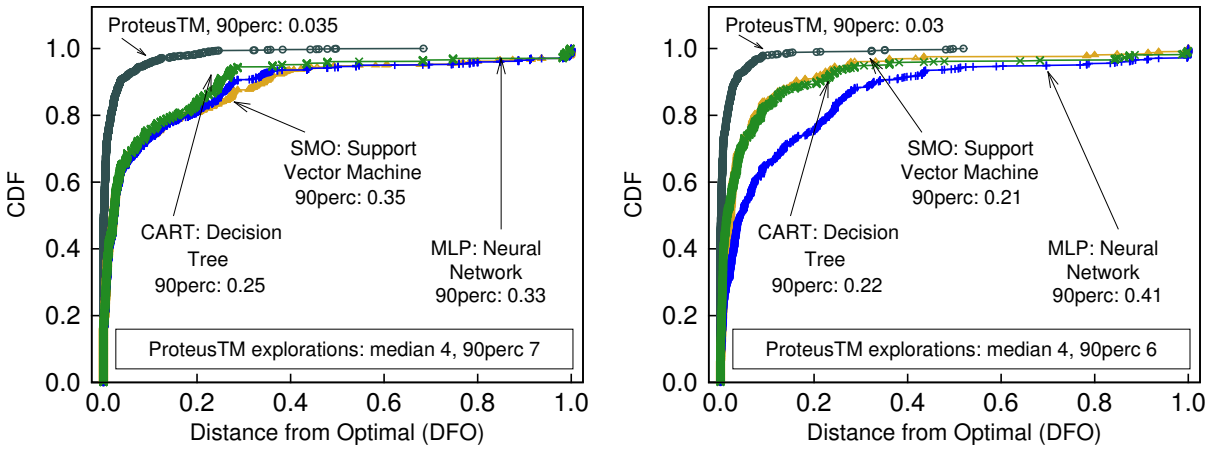
Figure 7.6: Comparison for the accuracy of different early-stop exploration predicates.

Stopping criterion. We now evaluate our stopping heuristic (recall Section 7.6.2), called *Cautious* (*C* in the plots). We compare it with a *Naive* stopping rule *N* that blindly trusts the model, by stopping explorations when the expected improvement over the best known configuration falls $< \epsilon$. The results are shown in Figure 7.6, portraying the sensitivity of both heuristics to value of ϵ.

For any fixed ϵ, we observe that the Naive predicate chooses consistently a worse configuration than the one of our Cautious heuristic: blindly trusting the predictive model results in an excessively eager policy, which does not provide the model with enough training data to achieve adequate accuracy as the model may be not receive enough data to learn the configuration-to-KPI relation for the workload.

As expected, the plots also show that the lower ϵ, the lower the obtained MDFO. Notably, for $\epsilon = 0.01$, the Controller achieves, in 90% of the cases, MDFO of only 12% and 5% for these different tests. This comes at the price of a higher number of explorations. We note that the Controller is able to keep this price very low, by requiring, on average, a similar number of explorations of a policy that performs a fixed amount of explorations and is tuned to deliver the same mean performance. This confirms the effectiveness of the Controller in wisely determining the duration of the profiling phase, by striking a balanced trade-off between the extent of on-line exploration and final performance.

Comparison with ML approaches. We now compare ProteusTM with the AutoTM approach [Wang et al., 2012b], described in Section 7.3.2, to automate the choice of the TM



(a) 30% of the training set used.

(b) 70% of the training set used.

Figure 7.7: Comparison of the learning approach of ProteusTM (embedded in RecTM) against several Machine Learning based techniques.

algorithm for a given workload. This approach relies on workload characterization data to train a ML-based classifier that is used to predict the best TM configuration for a given workload. The workload characterization uses 17 features: e.g., duration of transactions, data access patterns, and level of data contention. Wang et al. also use static analysis to obtain other features, e.g., the number of atomic blocks. We did not perform this step but complemented the features with contention management features. These are not considered by the authors, but we found them to be highly correlated with performance.

The simulation for ProteusTM evolves as previously explained. For the ML competitor, instead, a workload is first profiled over a reference configuration (TinySTM, 4 threads) to collect all the aforementioned features and then the ML is invoked to predict the best configuration. We then compute the MDFO for this predicted configuration.

We used the 300 STAMP and data-structures workloads as training set and split them randomly into training and test sets: 30-70 and 70-30 train-test splits. For ProteusTM, the training set is the UM corresponding to the selected workloads. For ML approaches, the training set is composed, for each workload, by the aforementioned features and the identifier of the best configuration as target class. We report these experiments on Machine A.

We consider 3 ML algorithms, implemented in Weka [Hall et al., 2009]: Decision Trees (CART), Support Vector Machines (SMO), and Artificial Neural Networks (MLP) [Bishop, 2006]. Their parameters were chosen via random search optimization [Bergstra and Bengio, 2012], which

evaluated 100 combinations with cross-validation on the training set.

Figure 7.7 reports the CDF of the DFO of each technique. The data shows the superiority of ProteusTM relatively to pure ML approaches. In particular, with 30% training set, ProteusTM already delivers a DFO of 1.6% against the 10% of the ML competitors, and a 90^{th} percentile of 3.5% against 25% of CART (the best alternative). Also, by increasing the training set to 70%, ProteusTM delivers a DFO of 1.3% and a 90^{th} percentile of 3%, against 6.8% DFO and 21% of the best alternative (SMO).

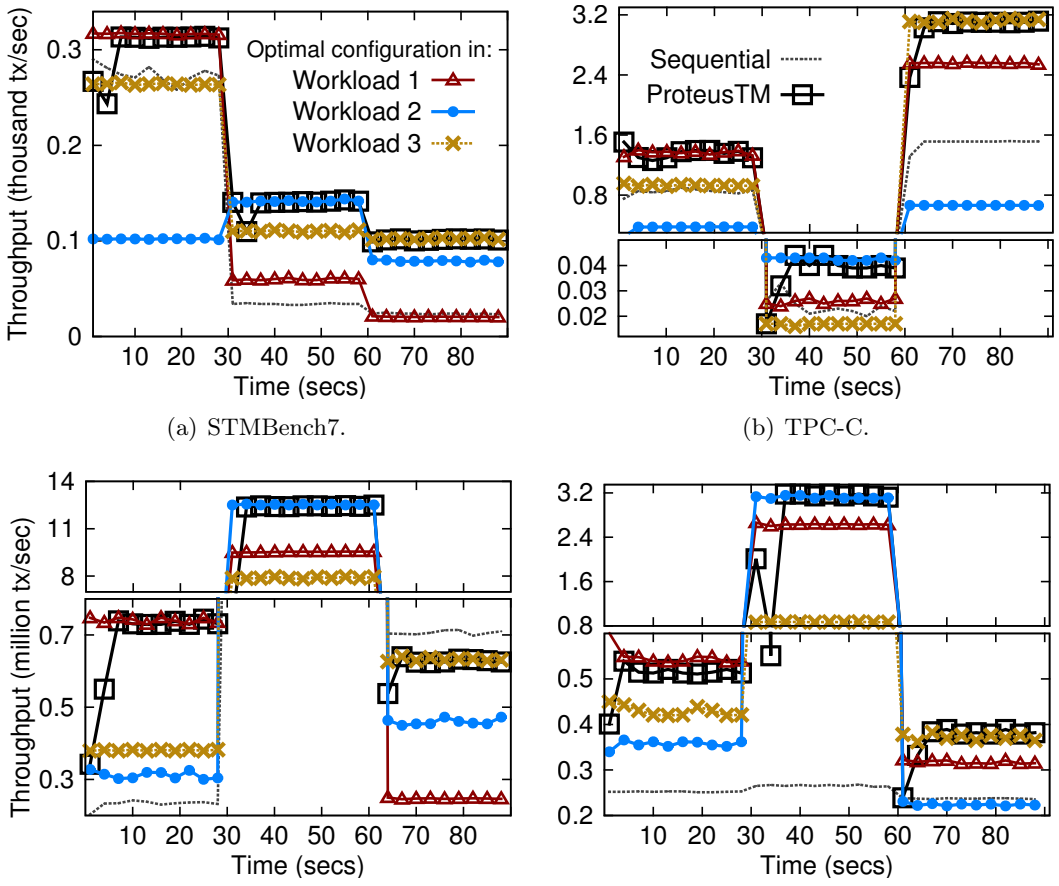
We note that the DFO of ProteusTM is similar (both in mean and 90^{th} percentile) in both cases, whereas ML greatly benefits from more training data. This difference can be explained by the number of explorations required by ProteusTM to perform its profiling phase (with threshold $\epsilon = 0.01$): at 30% training, the 90^{th} percentile number of explorations is 7, but this lowers to 6 with 70% training set. This means that ProteusTM delivers high accuracy also in presence of scarce training data, by autonomously exploring more.

Our evaluation suggests that detecting similarities on the KPI is more effective than statistically inferring relationships from training data. We argue that this depends on two, tightly intertwined, causes: 1) thanks to our novel normalization, using CF is more robust than ML, as it is based on *direct KPI observations*, rather than on *learning* the mapping of input to output features; and 2) the adaptive profiling phase proved to be more effective than a *one-shot* classification-based solution.

7.7.4 Online Optimization of Dynamic Workloads

In Figure 7.8, we evaluate the ProteusTM system as a whole. For each application we trigger 3 workloads chosen to exemplify contrasting characteristics and resulting performances. We stress that, in each case, ProteusTM is totally oblivious of the target application: no workloads of the application are present in its training set. This highlights the Recommender's ability to detect similarity patterns between the target workloads and the set of *disjoint* applications used as training set.

We set the Monitor period to 1 sec and the SMBO ϵ to 0.01. In each run, we measure the performance of 1) ProteusTM, 2) the 3 configurations that perform best in each workload, 3) the Best Fixed configuration on Average (BFA) across the three workloads of each application,



(a) STMBench7.

(b) TPC-C.

(c) Red-Black Tree (RBT).

(d) Memcached.

Figure 7.8: Performance of four applications when their workload changes three times over a period of 90 seconds. We show the performance obtained with ProteusTM, and three additional fixed configurations, each one corresponding to the best performing configuration in each of the three workloads.

and 4) a Sequential non-instrumented execution.

We draw three conclusions from these plots:

- ProteusTM is able to quickly identify, at runtime, configurations that are optimal — or very close. Remarkably, ProteusTM delivers performance that is, on average, only 1% lower than the optimal.
- Employing *any* of the baseline alternatives yields up to two orders of magnitude lower performance.
- Thanks to our SMBO approach, the performance degradation incurred when exploring is

Table 7.6: For each benchmark (of Figure 7.8), we show the MDFO (in %) of all the alternatives tested: ProteusTM, each of the optimal configurations in each of the three workloads for each application (denoted as Opt i for the i^{th} workload in the application), and the Best Fixed on Average (BFA) across the workloads of a given application (which always coincides with one of the optimal configurations, denoted with ★). Each workload is labelled with the specification of its optimal configuration (of which one is also the BFA, as mentioned previously). For the case of ProteusTM, we show also the number of explorations that it performed before deciding which configuration to use (on the right column, next to its MDFO).

Machine			**Mean Distance from Optimum (MDFO %)**			
	Benchmark		**Optimal in Workload i**			**ProteusTM**
	Name	**Workload (Opt Conf)**	**Opt 1**	**Opt 2**	**Opt 3**	**(explorations)**
A	RBT	1 (NOrec: 7t)	0	137	93	$<$ **1** (4 expl)
		2 ★ (HTM:8t Half-20)	33	0	71	**2** (4 expl)
		3 (HTM: 4t GiveUp-4)	154	37	0	$<$ **1** (7 expl)
A	STMB7	1 (HTM: 4t Linear-2)	0	20	210	**2** (6 expl)
		2 (Swiss: 4t)	135	0	28	$<$ **1** (4 expl)
		3 ★ (TL2: 8t)	390	29	0	$<$ **1** (3 expl)
A	TPC-C	1 ★ (Tiny: 4t)	0	273	47	$<$ **1** (3 expl)
		2(HTM:3t GiveUp-16)	68	0	152	**3** (4 expl)
		3 (Tiny: 8t)	22	370	0	$<$ **1** (3 expl)
B	Memchd	1 ★ (Swiss: 32t)	0	50	26	**4** (3 expl)
		2 (Tiny: 32t)	19	0	258	$<$ **1** (4 expl)
		3 (Tiny: 4t)	18	66	0	$<$ **1** (3 expl)

minimal (at most 7 explorations in these use cases). Such cost is usually amortized in long-running services (e.g., databases), in which workload shifts are infrequent [Curino et al., 2011].

A summary is provided in Table 7.6 where we list the optimal configurations in each workload. We also show the Best Fixed on Average (with ★) which is always also an optimal configuration in some workload. This data highlights the robustness of ProteusTM to optimize heterogeneous applications with diverse optimal configurations, in terms of TM algorithm (STMB7), parallelism degree (TPC-C) and HTM tuning (RBT and Memchd) across the two different machines.

Finally, in Figure 7.9, we confirm our claims of Section 7.6.3 by using a static TPC-C workload and varying external factors to the application to trigger behavior changes. As such, we do not change the workload of the application *per se*, but, instead, vary the resources available to the machine. To simulate these external changes we used the *stress* Unix tool with different

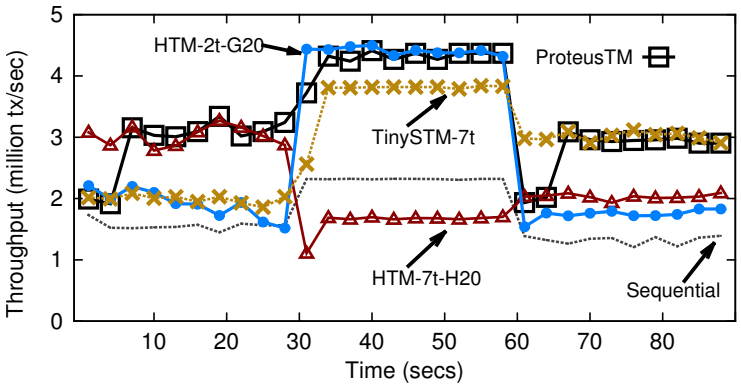


Figure 7.9: Similar to Figure 7.8(b), but with a static application workload, varying instead the availability of machine resources.

configurations over periods of 30 seconds: it either created high CPU, memory or IO usage in each workload. The results are similar to what was chosen previously, in that ProteusTM obtains perform close to the optimal across the test.

7.8 Summary

We proposed ProteusTM, the first TM system with multi-dimensional self-tuning capabilities. ProteusTM is integrated with GCC and exposes a standard TM interface, which ensures full transparency, ease of use and portability. Together with this simple abstraction, we provide high performance by relying on a novel optimization technique that leverages Collaborative Filtering and Bayesian Optimization.

Via an extensive evaluation based on real-word application and well-known benchmarks, we demonstrated the ability ProteusTM to optimize heterogeneous applications in high-dimensional configuration spaces: ProteusTM achieves performance that are, on average, $< 3\%$ from optimum and gains up to $100\times$ relatively to static configurations. These results confirm the high performance robustness delivered by ProteusTM.

This methodology allows to optimize several dimensions — namely, the best TM algorithm, configuration, and number of threads — and to seamlessly cope with possibly varying workloads,, while the software developer remains concerned only with writing her concurrent application with atomic blocks. This allows her to benefit from the trade-offs exhibited by STMs and HTMs,

which make them optimal under very different scenarios. This ability is particularly relevant in the light of the results of our study in Chapter 3, which highlighted that current HyTM systems deliver quite disappointing performance. It is also relevant to highlight that the solution conveyed with ProteusTM is possible thanks to the flexible abstraction of TM, which allows significantly different implementations to be used, all under the same interface.

8 Final Remarks

Transactional Memory (TM) was one of those ideas that may be said to have been proposed ahead of its time. At the time of the inception of TM, in the early nineties, concurrent programming was only known and used by a niche of experts. As such, the proposal of this new paradigm did not immediately trigger much attention. Years later, the ubiquitous adoption of multi-core processors coincided with a renewed interest in the paradigm of TM for concurrent programming. In less than a decade, this has evolved into the current situation in which mainstream compilers support programs written with TM constructs, and commodity processors have support for hardware TM execution. These notable advances are strongly motivated by the simplicity that TM allows in the development of concurrent programs.

While the abstraction for programming with TM has not changed much over the years, and indeed was acknowledged as effective given its wide adoption, the same cannot be said with respect to the performance of TM implementations. Much of the research effort in the past years has been focused on the performance side of TMs, driven by the frequent debate over the inconsistent performance of existing algorithms, whose implementations are known to have widely varying performance depending on the workloads and applications in which they are used.

This dissertation addresses exactly this problem of creating robust TM algorithms that can deliver high performance in presence of heterogeneous workloads, while preserving the simplicity of the TM abstraction. We have pursued this objective by proposing several new algorithms and self-tuning techniques for TM systems, which we evaluated in comparison with the state of the art to demonstrate significant performance improvements.

In more detail, we first conducted a large comparative study between various TMs and also locking schemes. Remarkably, this was one of the first (if not the first) work to conduct experimental performance study contrasting the performance of various state of the art STM and HyTM systems with the HTM implementation recently introduced by Intel for its commodity processors. Contrarily to the intuition that HTM would nullify the relevance of Software TM

(STM), our initial study demonstrated that the hardware limitations inherent to commodity HTMs, in fact, lead to a situation in which no one size fits all: both HTM and STM perform optimally in different types of workloads and applications. Furthermore, this initial step allowed us also to identify several performance issues with existing TMs, which we addressed in the scope of this dissertation.

In the scope of STMs, we devised the novel TIME-WARP algorithm that reduces the chance for transactions to be aborted, by using a different set of rules in the concurrency control performed. The key idea of TIME-WARP is to allow update transactions to execute as if they had already happened in the past, and thus avoiding conflicts with concurrent transactions, which would cause aborts and restarts.

We then focused on the HTM support available in recent processors, for which we identified two main problems.

On one hand, we observed that there was no one size fits all solution for configuring the software logic that governs the usage of the HTM. We filled this gap by introducing TUNER, an online self-tuning approach that manages the trade-off between exploring new configurations and adopting the best one seen so far. As such, this allows the programmer to remain oblivious of the limitations of HTMs and let our runtime configure it properly.

On the other hand, we tackled the problem of how to reduce the chance of transaction aborts also with HTMs. A key challenge inherent to addressing this problem is that, due to the complexity of modern CPUs, processor manufactures are extremely conservative in changing the logic placed in the processors' micro-code. Rather than introducing an alternative concurrency control mechanism (as was the case with TIME-WARP, which targeted STMs), we proposed the first (software) scheduling algorithm, SEER, designed to cope with the restrictions of currently available HTM implementations.

Finally, we note that we had also highlighted the fact that both STMs and HTMs can perform optimally in different scenarios. So, in fact, it is desirable to use both depending on the current characteristics of the workload. Unfortunately, it is undesirable to expose these intricacies to the programmer, as it would be complex for her to devise rules and a switching system to allow adapting between different implementations. This is exactly what we proposed to achieve automatically with PROTEUSTM: an adaptive TM system, which monitors a performance indicator at runtime, and then strives to deliver the best possible performance by transparently

adapting not only the underlying TM implementation, but also several configurations parameters (such as the number of active threads, the configuration of the retry policy for HTMs, and contention management between conflicting transactions).

The common traits of all these contributions are noteworthy performance improvements over the state of the art TM algorithms, which were achieved while preserving the simplicity and elegance of the TM abstraction.

The need to adopt adaptive solutions has been identified already in the first contribution of this dissertation, which unveiled several situations in which contrasting TMs would outperform each other, depending on the workloads' characteristics. An important aspect of the design choices that we adopted, while devising the proposed TM self-tuning schemes, was the reliance on online learning techniques. These have the notable advantage of not requiring the programmer to provide any *a priori* information about the workload. Hence, sparing her from any additional sources of complexity and achieving full transparency. Furthermore, it was of the utmost relevance to adopt self-tuning designs based on very lightweight and efficient mechanisms, as otherwise we would easily incur in situations in which the gains would be outweigh by the overheads of the ability to adapt.

The work shown here also demonstrated that the available Hybrid TM (HyTM) approaches tend to perform sub-optimally with respect to the case in which one can choose between various HTMs and STMs depending on the workload. Although manually perform such adaptation is not practical, this was made possible with the contribution of ProteusTM that automatizes that choice (among others). Coincidently with our practical experience, a recent theoretical contribution has proven the inherent high costs of HyTMs in terms of their overheads and associated software instrumentation in the hardware transactions [Alistarh et al., 2015]. In this sense, it is an interesting research direction to study further these inherent limitations, or else to show practically that one can build a better HyTM that could perform closely to the best pure approaches in different scenarios. A key aspect of this may be to consider the availability of non-transactional operations in the HTM implementations. These were not studied theoretically in the inherent limitations mentioned above, albeit we hypothesize that they may prove to be relevant in terms of the upper bounds shown for the overhead costs. Such studies could also help to motivate the processors' manufacturers to include non-transactional accesses in future releases of their HTMs.

Another interesting research avenue for expanding the effectiveness of TM systems is to consider Non-Uniform Memory Access (NUMA) machines. The underlying architectures of these systems dictates that the cost of accessing memory, in terms of latency, varies depending on the processor core and memory bank being accessed. As a result, the management of threads — in terms of their mapping to cores [Castro et al., 2014] — as well as the data management — in terms of its locality [Dice et al., 2012b] — turn out to be similar challenges to those present in distributed systems [Sowell et al., 2012, Diegues and Romano, 2015b]. The amount of work available in the TM literature for NUMA systems is rather small (e.g., a NUMA-aware TM was published only very recently [Mohamedin et al.,]), when considering the trending relevance of these machine architectures, and even more given that it is the most likely way that such machines will have to expand the ever-growing number of processor cores. As such, it is of the utmost importance to explore adaptive ideas and novel algorithms that provide robust performance for TM systems also in NUMA machines.

References

Adir, A., Goodman, D., Hershcovich, D., Hershkovitz, O., Hickerson, B., Holtz, K., Kadry, W., Koyfman, A., Ludden, J., Meissner, C., Nahir, A., Pratt, R. R., Schiffli, M., St. Onge, B., Thompto, B., Tsanko, E., and Ziv, A. (2014). Verification of transactional memory in power8. In *Proceedings of the Annual Design Automation Conference*, DAC, pages 58:1–58:6.

Adl-Tabatabai, A., Shpeisman, T., and Gottschlich, J. (2012). Draft Specification of Transactional Language Constructs for C++. In *Intel*.

Adya, A. (1999). *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology.

Afek, Y., Levy, A., and Morrison, A. (2013). Programming with hardware lock elision. In *Procceedings of Principles and Practice of Parallel Programming*, PPoPP, pages 295–296.

Afek, Y., Levy, A., and Morrison, A. (2014). Software-Improved Hardware Lock Elision. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC.

Alistarh, D., Kopinsky, J., Kuznetsov, P., Ravi, S., and Shavit, N. (2015). Inherent Limitations of Hybrid Transactional Memory. In *In Proceedings of the International Conference on Distributed Computing*, pages 185–199.

Ananian, C. S., Asanovic, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S. (2006). Unbounded transactional memory. *IEEE Micro*, 26(1):59–69.

Ansari, M., Khan, B., Luján, M., Kotselidis, C., Kirkham, C. C., and Watson, I. (2010). Improving Performance by Reducing Aborts in Hardware Transactional Memory. In *Proceedings of the Conference on High Performance Embedded Architectures and Compilers*, HiPEAC, pages 35–49.

Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C. C., and Watson, I. (2009). Steal-on-Abort: Improving Transactional Memory Performance Through Dynamic Transaction Reordering. In *Proceedings on the Conference on High Performance Embedded Architectures*

and Compilers, HiPEAC, pages 4–18.

Attiya, H. and Hillel, E. (2011). Single-version STMs can be multi-version permissive. In *Proceedings of the International Conference on Distributed Computing and Networking*, ICDCN, pages 83–94.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Journal of Machine Learning*, 47(2-3):235–256.

Aydonat, U. and Abdelrahman, T. (2012). Relaxed Concurrency Control in Software Transactional Memory. *Transactions on Parallel and Distributed Systems (TPDS)*, 23(7):1312–1325.

Baldassin, A., Klein, F., Araujo, G., Azevedo, R., and Centoducatte, P. (2009). Characterizing the Energy Consumption of Software Transactional Memory. *IEEE Computer Architecture Letters*, 8(2):56–59.

Basseville, M. and Nikiforov, I. V. (1993). *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. (1995). A critique of ANSI SQL isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–10.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for Hyper-Parameter Optimization. In *Proceedings of the Conference on Neural Information Processing Systems*, volume 24 of *NIPS*.

Bergstra, J. and Bengio, Y. (2012). Random Search for Hyper-parameter Optimization. *Journal of Machine Learning Research*, 13(1):281–305.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.

Brochu, E., Cora, V. M., and de Freitas, N. (2010). A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. eprint arXiv:1012.2599, arXiv.org.

Cahill, M. J., Röhm, U., and Fekete, A. D. (2008). Serializable Isolation for Snapshot Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD, pages 729–738.

Cain, H., Michael, M., Frey, B., May, C., Williams, D., and Le, H. (2013). Robust architectural support for transactional memory in the power architecture. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 225–236.

Calciu, I., Gottschlich, J., Shpeisman, T., Pokam, G., and Herlihy, M. (2014a). Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 187–200.

Calciu, I., Shpeisman, T., Pokam, G., and Herlihy, M. (2014b). Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory. In *Workshop on Transactional Computing*, TRANSACT.

Carey, M. J., DeWitt, D. J., and Naughton, J. F. (1993). The 007 Benchmark. *SIGMOD Record*, 22(2):12–21.

Carvalho, N., Romano, P., and Rodrigues, L. (2010). Asynchronous lease-based replication of software transactional memory. In *Proceedings of Middleware*, pages 376–396.

Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S., and Chatterjee, S. (2008). Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6(5):40:46–40:58.

Castro, M., Góes, L. F. W., and Méhaut, J.-F. (2014). Adaptive Thread Mapping Strategies for Transactional Memory Applications. *Journal of Parallel and Distributed Computing*, 74(9):2845–2859.

Christie, D., Chung, J.-W., Diestelhorst, S., Hohmuth, M., Pohlack, M., Fetzer, C., Nowack, M., Riegel, T., Felber, P., Marlier, P., and Rivière, E. (2010). Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the European Conference on Computer Systems*, EuroSys, pages 27–40.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of Symposium on Cloud Computing*, SoCC, pages 143–154.

Couceiro, M., Romano, P., Carvalho, N., and Rodrigues, L. (2009). D2STM: Dependable Distributed Software Transactional Memory. In *Proceedings of the Pacific Rim Symposium on Dependable Computing*, PRDC, pages 307–313.

Crain, T., Imbs, D., and Raynal, M. (2011). Read invisibility, virtual world consistency and probabilistic permissiveness are compatible. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP, pages 244–257.

Curino, C., Jones, E. P., Madden, S., and Balakrishnan, H. (2011). Workload-aware database monitoring and consolidation. In *Proceedings of the International Conference on Management of Data*, SIGMOD, pages 313–324.

Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M. L., and Spear, M. F. (2011). Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 39–52.

Dalessandro, L., Spear, M. F., and Scott, M. L. (2010). NOrec: streamlining STM by abolishing ownership records. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 67–78.

Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. (2006). Hybrid Transactional Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 336–346.

Das, A. S., Datar, M., Garg, A., and Rajaram, S. (2007). Google News Personalization: Scalable Online Collaborative Filtering. In *Proceedings of the International Conference on World Wide Web*, WWW, pages 271–280.

David, H., Gorbatov, E., Hanebutte, U., Khanna, R., and Le, C. (2010). RAPL: memory power estimation and capping. In *Proceedings of the International Symposium on Low power Electronics and Design*, ISLPED, pages 189–194.

David, T., Guerraoui, R., and Trigonakis, V. (2013). Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of Symposium on Operating Systems Principles*, SOSP, pages 33–48.

Davidson, J., Liebald, B., Liu, J., Nandy, P., Van Vleet, T., Gargi, U., Gupta, S., He, Y., Lambert, M., Livingston, B., and Sampath, D. (2010). The YouTube Video Recommen-

dation System. In *Proceedings of the Conference on Recommender Systems*, RecSys, pages 293–296.

Delimitrou, C. and Kozyrakis, C. (2013). Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 77–88.

Delimitrou, C. and Kozyrakis, C. (2014). Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 127–144.

Di Sanzo, P., Ciciani, B., Palmieri, R., Quaglia, F., and Romano, P. (2012). On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking. *Perform. Eval.*, 69(5):187–205.

Dice, D., Harris, T., Kogan, A., Lev, Y., and Moir, M. (2014a). Pitfalls of lazy subscription. In *Workshop on the Theory of Transactional Memory*, WTTM.

Dice, D., Herlihy, M., Lea, D., Lev, Y., Luchangco, V., Mesard, W., Moir, M., Moore, K., and Nussbaum, D. (2008). Applications of the Adaptive Transactional Memory Test Platform. In *3rd Workshop on Transactional Computing*, TRANSACT.

Dice, D., Kogan, A., Lev, Y., Merrifield, T., and Moir, M. (2014b). Adaptive Integration of Hardware and Software Lock Elision Techniques. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 188–197.

Dice, D., Lev, Y., Moir, M., and Nussbaum, D. (2012a). Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 157–168.

Dice, D., Marathe, V. J., and Shavit, N. (2012b). Lock cohorting: a general technique for designing NUMA locks. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 247–256.

Dice, D., Shalev, O., and Shavit, N. (2006). Transactional Locking II. In *Proceedings of the Symposium on Distributed Computing*, DISC, pages 194–208.

Didona, D., Felber, P., Harmanci, D., Romano, P., and Schenker, J. (2013). Identifying the Optimal Level of Parallelism in Transactional Memory Applications. In *Proceedings of the Inernational Conference on Networked Systems*, NETYS, pages 233–247.

Didona, D., Romano, P., Peluso, S., and Quaglia, F. (2014). Transactional Auto Scaler: Elastic Scaling of Replicated In-Memory Transactional Data Grids. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(2):11:1–11:32.

Diegues, N. and Cachopo, J. (2013). Practical Parallel Nesting for Software Transactional Memory. In *Proceedings of the Symposium on Distributed Computing*, DISC, pages 149–163.

Diegues, N. and Romano, P. (2015a). Self-Tuning Intel Restricted Transactional Memory. *Elsevier Parallel Computing*, pages 25–52.

Diegues, N. and Romano, P. (2015b). STI-BT: A Scalable Transactional Index. *IEEE Transactions on Parallel Distributed Systems (TPDS)*.

Dolev, S., Hendler, D., and Suissa, A. (2008). CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC, pages 125–134.

Dragojević, A., Felber, P., Gramoli, V., and Guerraoui, R. (2011). Why STM Can Be More Than a Research Toy. *Communications of ACM*, 54(4):70–77.

Dragojević, A., Guerraoui, R., and Kapalka, M. (2009a). Stretching transactional memory. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 155–165.

Dragojević, A., Guerraoui, R., Singh, A. V., and Singh, V. (2009b). Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC, pages 7–16.

Felber, P., Fetzer, C., Marlier, P., and Riegel, T. (2010a). Time-Based Software Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807.

Felber, P., Fetzer, C., and Riegel, T. (2008). Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 237–246.

Felber, P., Issa, S., Matveev, A., and Romano, P. (2016). Hardware read-write lock elision. In *Proceedings of the European Conference on Computer Systems*, EuroSys, page 34:1–34:15.

Felber, P., Korland, G., and Shavit, N. (2010b). Deuce: Noninvasive concurrency with a Java STM. In *Proceedings of the Workshop on Programmability Issues for Multi-Core Computers*,

MULTIPROG.

Fernandes, S. M. and Cachopo, J. (2011). Lock-free and scalable multi-version software transactional memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 179–188.

Ferri, C., Bahar, R. I., Loghi, M., and Poncino, M. (2009). Energy-optimal Synchronization Primitives for Single-chip Multi-processors. In *Proceedings of the Great Lakes Symposium on VLSI*, GLSVLSI, pages 141–144.

Ferri, C., Wood, S., Moreshet, T., Iris Bahar, R., and Herlihy, M. (2010). Embedded-TM: Energy and complexity-effective hardware transactional memory for embedded multicore systems. *Journal of Parallel and Distributed Computing*, 70(10):1042–1052.

Gaona, E., Titos, R., Fernandez, J., and Acacio, M. (2013). On the design of energy-efficient hardware transactional memory systems. *Concurrency and Computation: Practice and Experience*, 25(6):862–880.

Garivier, A. and Moulines, E. (2011). On Upper-confidence Bound Policies for Switching Bandit Problems. In *Proceedings of the International Conference on Algorithmic Learning Theory*, ALT, pages 174–188.

Gautham, A., Korgaonkar, K., Slpsk, P., Balachandran, S., and Veezhinathan, K. (2012). The implications of shared data synchronization techniques on multi-core energy efficiency. In *Proceedings of Power-Aware Computing and Systems*, HotPower, pages 1–6.

Goel, B., Gil, J. R. T., Negi, A., McKee, S. A., and Stenström, P. (2014). Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 615–624.

Gramoli, V. (2015). More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 1–10.

Gramoli, V., Harmanci, D., and Felber, P. (2010). On the Input Acceptance of Transactional Memory. *Parallel Processing Letters*, 20(1).

Guerraoui, R., Henzinger, T. A., and Singh, V. (2008). Permissiveness in Transactional Mem-

ories. In *Proceedings of the Symposium on Distributed Computing*, DISC, pages 305–319.

Guerraoui, R., Herlihy, M., and Pochon, B. (2005a). Polymorphic contention management. In *Proceedings of the Symposium on Distributed Computing*, DISC, pages 303–323.

Guerraoui, R., Herlihy, M., and Pochon, B. (2005b). Toward a Theory of Transactional Contention Managers. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC, pages 258–264.

Guerraoui, R. and Kapalka, M. (2008). On the correctness of transactional memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 175–184.

Guerraoui, R., Kapalka, M., and Vitek, J. (2007). STMBench7: a benchmark for software transactional memory. In *Proceedings of the European Conference on Computer Systems*, pages 315–324.

Hackenberg, D., Ilsche, T., Schone, R., Molka, D., Schmidt, M., and Nagel, W. (2013). Power measurement techniques on standard compute nodes: A quantitative comparison. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 194–204.

Hähnel, M., Döbel, B., Völp, M., and Härtig, H. (2012). Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Performance Evaluation Review*, 40(3):13–17.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1):10–18.

Hammond, L., Wong, V., Chen, M., Carlstrom, B., Davis, J., Hertzberg, B., Prabhu, M., Wijaya, H., Kozyrakis, C., and Olukotun, K. (2004). Transactional Memory Coherence and Consistency. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 102–114.

Herlihy, M. (1991). Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149.

Herlihy, M. and Koskinen, E. (2008). Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 207–216.

Herlihy, M., Luchangco, V., and Moir, M. (2003a). Obstruction-Free Synchronization: Double-Ended Queues As an Example. In *Proceedings of the International Conference on Distributed Computing Systems*, ICDCS, pages 522–529.

Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N. (2003b). Software transactional memory for dynamic-sized data structures. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC, pages 92–101.

Herlihy, M. and Moss, J. E. B. (1993). Transactional Memory: architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 289–300.

Herlihy, M. and Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Hirve, S., Palmieri, R., and Ravindran, B. (2014). HiperTM: High Performance, Fault-Tolerant Transactional Memory. In *Proceedings of the International Conference on Distributed Computing and Networking*, ICDCN, pages 181–196.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Sequential Model-based Optimization for General Algorithm Configuration. In *Proceedings of the International Conference on Learning and Intelligent Optimization*, LION, pages 507–523.

Imbs, D. and Raynal, M. (2012). Virtual World Consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science*, 444(0):113–127.

Intel Corporation (2009). Intel Transactional Memory Compiler and Runtime Application Binary Interface. https://gcc.gnu.org/wiki/TransactionalMemory?action=AttachFile&do=get&target=Intel-TM-ABI-1_1_20060506.pdf.

Intel Corporation (February 2012). Intel Architecture Instruction Set Extensions Programming Reference. In *319433-012 edition*.

Issa, S., Romano, P., and Brorsson, M. (2015). Green-CM: Energy Efficient Contention Management for Transactional Memory. In *Proceedings of the International Conference on Parallel Processing*, ICPP, pages 550–559.

Jacobi, C., Slegel, T., and Greiner, D. (2012). Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the Symposium on Microarchitecture*,

MICRO, pages 25–36.

Jefferson, D. R. (1985). Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425.

Jones, D. R., Schonlau, M., and Welch, W. J. (1998). Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13(4):455–492.

Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. (2008). H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Journal of the VLDB Endowment*, 1(2):1496–1499.

Karnagel, T., Dementiev, R., Rajwar, R., Lai, K., Legler, T., Schlegel, B., and Lehner, W. (2014). Improving In-Memory Database Index Performance with Intel TSX. In *Proc. of International Symposium on High Performance Computer Architecture*, HPCA, pages 476–487.

Kaufmann, E., Cappé, O., and Garivier, A. (2012). On Bayesian Upper Confidence Bounds for Bandit Problems. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, volume 22, pages 592–600.

Keidar, I. and Perelman, D. (2009). On avoiding spare aborts in transactional memory. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 59–68.

Kleen, A. (2014). Scaling existing lock-based applications with lock elision. *Queue*, 12(1):20:20–20:27.

Kobus, T., Kokocinski, M., and Wojciechowski, P. T. (2013). Hybrid Replication: State-Machine-Based and Deferred-Update Replication Schemes Combined. In *Proceedings of the International Conference on Distributed Computing Systems*, ICDCS, pages 286–296.

Kotselidis, C., Luján, M., Ansari, M., Malakasis, K., Khan, B., Kirkham, C. C., and Watson, I. (2010). Clustering JVMs with software transactional memory support. In *Proceedings of the International Parallel Distributed Processing Symposium*, IPDPS, pages 1–12.

Kuvaiskii, D., Faqeh, R., Bhatotia, P., Felber, P., and Fetzer, C. (2016). HAFT: Hardware-assisted Fault Tolerance. In *Proceedings of the European Conference on Computer Systems*, EuroSys, pages 25:1–25:17.

Larson, P.-A., Blanas, S., Diaconu, C., Freedman, C., Patel, J. M., and Zwilling, M. (2011). High-performance Concurrency Control Mechanisms for Main-memory Databases. *Proceedings of the VLDB Endownment Journal*, 5(4):298–309.

Le, H. Q., Guthrie, G. L., Williams, D. E., Michael, M. M., Frey, B. G., Starke, W. J., May, C., Odaira, R., and Nakaike, T. (2015). Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8:1–8:14.

Leung, K.-C., Chen, Y., and Huang, Z. (2013). Restricted admission control in view-oriented transactional memory. *Journal of Supercomputing*, 63(2):348–366.

Lev, Y., Moir, M., and Nussbaum, D. (2007). PhTM: Phased Transactional Memory. In *Workshop on Transactional Computing*, TRANSACT.

Li, X., Andersen, D. G., Kaminsky, M., and Freedman, M. J. (2014). Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the European Conference on Computer Systems*, EuroSys.

Linden, G., Smith, B., and York, J. (2003). Amazon.Com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing*, 7(1):76–80.

Lu, L. and Scott, M. L. (2013). Generic Multiversion STM. In *Proceedings of the Symposium on Distributed Computing*, DISC, pages 134–148.

Lupei, D., Simion, B., Pinto, D., Misler, M., Burcea, M., Krick, W., and Amza, C. (2010). Transactional Memory Support for Scalable and Transparent Parallelization of Multiplayer Games. In *Proceedings of the European Conference on Computer Systems*, EuroSys, pages 41–54.

Maldonado, W., Marlier, P., Felber, P., Suissa, A., Hendler, D., Fedorova, A., Lawall, J. L., and Muller, G. (2010). Scheduling Support for Transactional Memory Contention Management. In *Proceedings of Principles and Practice of Parallel Programming*, PPoPP, pages 79–90.

Mannarswamy, S., Chakrabarti, D., Rajan, K., and Saraswati, S. (2010). Compiler aided selective lock assignment for improving the performance of software transactional memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 37–46.

Marathe, V. J., Spear, M. F., and Scott, M. L. (2008). Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *Proceedings of the International*

Conference on Parallel Processing, ICPP, pages 67–74.

Matveev, A. and Shavit, N. (2013). Reduced hardware transactions: a new approach to hybrid transactional memory. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 11–22.

Matveev, A. and Shavit, N. (2015). Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 59–71.

McDonald, A., Chung, J., Chafi, H., Minh, C. C., Carlstrom, B. D., Hammond, L., Kozyrakis, C., and Olukotun, K. (2005). Characterization of TCC on Chip-Multiprocessors. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 63–74.

Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K. (2008). STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the Symposium on Workload Characterization*, IISWC, pages 35–46.

Mohamedin, M., Palmieri, R., Hassan, A., and Ravindran, B. (2015a). Brief Announcement: Managing Resource Limitation of Best-Effort HTM. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 71–73.

Mohamedin, M., Palmieri, R., Peluso, S., and Ravindran, B.

Mohamedin, M., Palmieri, R., and Ravindran, B. (2015b). Brief Announcement: On Scheduling Best-Effort HTM Transactions. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 74–76.

Morrison, A. and Afek, Y. (2013). Fast Concurrent Queues for x86 Processors. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 103–112.

Nakaike, T., Odaira, R., Gaudet, M., Michael, M. M., and Tomari, H. (2015). Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 144–157.

Ni, Y., Welc, A., Adl-Tabatabai, A.-R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., and Tian, X.

(2008). Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA, pages 195–212.

Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. (2013). Scaling Memcache at Facebook. In *Proceedings of the Conference on Networked Systems Design and Implementation*, NSDI, pages 385–398.

Odaira, R., Castanos, J., and Tomari, H. (2014). Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 131–142.

Osogami, T. and Kato, S. (2007). Optimizing System Configurations Quickly by Guessing at the Performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS, pages 145–156.

Owen, S., Anil, R., Dunning, T., and Friedman, E. (2011). *Mahout in Action*. Manning Publications Co., Greenwich, CT, USA.

Paiva, J., Ruivo, P., Romano, P., and Rodrigues, L. (2013). AutoPlacer: scalable self-tuning data placement in distributed key-value stores. In *Proceedings of the International Conference on Autonomic Computing*, ICAC, pages 119–131.

Pankratius, V. and Adl-Tabatabai, A.-R. (2011). A Study of Transactional Memory vs. Locks in Practice. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 43–52.

Papadimitriou, C. H. (1979). The serializability of concurrent database updates. *Journal of ACM*, 26(4):631–653.

Peluso, S., Fernandes, J., Romano, P., Quaglia, F., and Rodrigues, L. (2012a). SPECULA: Speculative Replication of Software Transactional Memory. In *Proceedings of the Symposium on Reliable and Distributed Systems*, SRDS, pages 91–100.

Peluso, S., Romano, P., and Quaglia, F. (2012b). SCORe: A Scalable One-copy Serializable Partial Replication Protocol. In *Proceedings of Middleware*, pages 456–475.

Peluso, S., Ruivo, P., Romano, P., Quaglia, F., and Rodrigues, L. (2012c). When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. In

Proceedings of the International Conference on Distributed Computing Systems, ICDCS, pages 455–465.

Perelman, D., Byshevsky, A., Litmanovich, O., and Keidar, I. (2011). SMV: Selective Multi-Versioning STM. In *Proceedings of the Symposium on Distributed Computing*, DISC, pages 125–140.

Perelman, D., Fan, R., and Keidar, I. (2010). On maintaining multiple versions in stm. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC, pages 16–25.

Pettijohn, E., Guo, Y., Lama, P., and Zhou, X. (2014). User-centric heterogeneity-aware mapreduce job provisioning in the public cloud. In *Proceedings of the International Conference on Autonomic Computing*, ICAC, pages 137–143.

Rajaraman, A. and Ullman, J. D. (2011). *Mining of Massive Datasets*. Cambridge University Press.

Rajwar, R. and Goodman, J. R. (2001). Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the Symposium on Microarchitecture*, MICRO, pages 294–305.

Ramadan, H. E., Roy, I., Herlihy, M., and Witchel, E. (2009). Committing conflicting transactions in an STM. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 163–172.

Riegel, T., Marlier, P., Nowack, M., Felber, P., and Fetzer, C. (2011). Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 53–64.

Rito, H. and Cachopo, J. (2014). ProPS: A Progressively Pessimistic Scheduler for Software Transactional Memory. In *Proc. European Conference on Parallel Processing*, Euro-Par, pages 150–161.

Ritson, C. G. and Barnes, F. R. M. (2013). An Evaluation of Intel's Restricted Transactional Memory for CPAs. In *Communicating Process Architectures*, CPA, pages 271–292.

Rossbach, C. J., Hofmann, O. S., Porter, D. E., Ramadan, H. E., Aditya, B., and Witchel, E. (2007). TxLinux: Using and Managing Hardware Transactional Memory in an Operating

System. In *Proceedings of Symposium on Operating Systems Principles*, SOSP, pages 87–102.

Rossbach, C. J., Hofmann, O. S., and Witchel, E. (2010). Is Transactional Programming Actually Easier? In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 47–56.

Rughetti, D., Di Sanzo, P., Ciciani, B., and Quaglia, F. (2012). Machine Learning-Based Self-Adjusting Concurrency in Software Transactional Memory Systems. In *Proceedings of Modeling, Analysis Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 278–285.

Rughetti, D., Romano, P., Quaglia, F., and Ciciani, B. (2014). Automatic Tuning of the Parallelism Degree in Hardware Transactional Memory. In *Proceedings of the European Conference on Parallel Processing*, Euro-Par, pages 475–486.

Ruivo, P., Couceiro, M., Romano, P., and Rodrigues, L. (2011). Exploiting Total Order Multicast in Weakly Consistent Transactional Caches. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, PRDC, pages 99–108.

Russell, S. J. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition.

Sanzo, P. D., Re, F. D., Rughetti, D., Ciciani, B., and Quaglia, F. (2013). Regulating concurrency in software transactional memory: An effective model-based approach. In *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems*, SASO, pages 31–40.

Schiper, N., Sutra, P., and Pedone, F. (2010). P-Store: Genuine Partial Replication in Wide Area Networks. In *Proceedings of Symposium on Reliable and Distributed Systems*, SRDS, pages 214–224.

Schneider, F. B. (1990). Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319.

Sciascia, D., Pedone, F., and Junqueira, F. (2012). Scalable Deferred Update Replication. In *Proceedings of the Conference on Dependable Systems and Networks*, DSN, pages 1–12.

Shavit, N. and Touitou, D. (1995). Software Transactional Memory. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC, pages 204–213.

Sonmez, N., Harris, T., Cristal, A., Unsal, O., and Valero, M. (2009). Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *Proceedings of the International Symposium on Parallel Distributed Processing*, IPDPS, pages 1–10.

Sowell, B., Golab, W., and Shah, M. A. (2012). Minuet: A Scalable Distributed Multiversion B-tree. *Proceedings of the VLDB Endowment*, 5(9):884–895.

Spear, M., Vyas, T., and Ruan, Wenjia Liu, Y. (2014). Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 399–412.

Spear, M. F., Marathe, V. J., Scherer, W. N., and Scott, M. L. (2006). Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proceedings of the International Conference on Distributed Computing*, DISC, pages 179–193.

Stone, J. M., Stone, H. S., Heidelberger, P., and Turek, J. (1993). Multiple reservations and the Oklahoma update. *IEEE Journal of Parallel Distributed Technology: Systems Applications*, 1(4):58–71.

Su, G. and Heisig, S. (2013). Experiences with Disjoint Data Structures in a New Hardware Transactional Memory System. In *Proceedings of the International Symposium Computer Architecture and High Performance Computing*, SBAC-PAD, pages 9–16.

Su, X. and Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in Artificial Intelligence*, 2009:4:2–4:2.

Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition.

Székely, G. J., Rizzo, M. L., and Bakirov, N. K. (2007). Measuring and testing dependence by correlation of distances. *The Annals of Statistics*, 35(6):2769–2794.

Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, KDD, pages 847–855.

TPC Council (2011). TPC-C Benchmark. http://www.tpc.org/tpcc.

Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. (2013). Speedy Transactions in

Multicore In-memory Databases. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 18–32.

Usui, T., Behrends, R., Evans, J., and Smaragdakis, Y. (2009). Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 3–14.

Vale, T., Dias, R., and Lourenço, J. (2013). On the Relevance of Total-Order Broadcast Implementations in Replicated Software Transactional Memories. In *Proceedings of the Conference on Multicore Software Engineering, Performance, and Tools*, MUSEPAT, pages 49–60.

Viktor Leis, Alfons Kemper, T. N. (2014). Exploiting Hardware Transactional Memory in Main-Memory Databases. In *Proceedings of the International Conference on Data Engineering*, ICDE, pages 580–591.

Wang, A., Gaudet, M., Wu, P., Amaral, J. N., Ohmacht, M., Barton, C., Silvera, R., and Michael, M. (2012a). Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 127–136.

Wang, Q., Kulkarni, S., Cavazos, J., and Spear, M. (2012b). A transactional memory with automatic performance tuning. *ACM Transactions on Architectures and Code Optimization (TACO)*, 8(4):54:1–54:23.

Wang, Z., Qian, H., Li, J., and Chen, H. (2014). Using Restricted Transactional Memory to Build a Scalable In-Memory Database. In *Proceedings of the European Conference on Computer Systems*, EuroSys.

Yen, L., Bobba, J., Marty, M. R., Moore, K. E., Volos, H., Hill, M. D., Swift, M. M., and Wood, D. A. (2007). LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proceedings of the International Symposium on High Performance Computer Architecture*, HPCA, pages 261–272.

Yoo, R. M., Hughes, C. J., Lai, K., and Rajwar, R. (2013). Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 1–11.

Yoo, R. M. and Lee, H.-H. S. (2008). Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 169–178.