On the Performance of Software Transactional Memory

THÈSE N^{O} 5386 (2012)

PRÉSENTÉE LE 1ER JUIN 2012

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE PROGRAMMATION DISTRIBUÉE

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Aleksandar DRAGOJEVIĆ

acceptée sur proposition du jury:

Prof. A. Schiper, président du jury
Prof. R. Guerraoui, directeur de thèse
Prof. P. Felber, rapporteur
Dr T. Harris, rapporteur
Prof. V. Kuncak, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

To Marija for all her love and support.

Acknowledgements

I would first like to thank my thesis supervisor Rachid Guerraoui without whose advice and guidance this thesis would not have been written. I also thank the president of the jury André Schiper and the members of the jury Tim Harris, Pascal Felber, and Viktor Kuncak for reading the preliminary version of the thesis and making sure that it is up to the high standards required by EPFL. I would also like to thank people from Intel, Oracle Labs, and Microsoft Research for giving me the opportunity to work with them during my summer internships, and improve the way I think about research problems and solve them. I thank all the people I worked with, talked with, and had lunches with, most of all: Yang Ni and Gilles Pokam from Intel, Mark Moir, Virendra Marathe, Victor Luchangco, and Yossi Lev from Oracle, and Tim Harris from Microsoft Research.

A big thank you goes to all former and current members of LPD lab at EPFL: my office mates Michał Kapałka and Radu Banabic, lab's secretary Kristine Verhamme, for helping with many non-technical issues which made my life much easier, lab's system administrator Fabien Salvi, for keeping the computers running smoothly allowing me to focus on my work, Vincent Gramoli, for the French version of thesis' abstract, Dan Alistarh, Nikola Knežević, Seth Gilbert, Marko Vukolić, Jesper Honig Spring, Maxime Monod, Florian Huc, Ron Levy, Victor Bushkov, Mihai Letia, Giuliano Losa, Vasileios Trigonakis, Waheed Ghumman, and Maysam Yabandeh.

I would like to thank all the friends in Lausanne, most of all: Nedeljko Vasić, Nikola Knežević, Mihailo Kolundžija, Matthieu Guerquin-Kern, Miloš Stanisavljević, Dejan Novaković, Sergey Korovnikov, Alexander Sennhauser for providing many hours of insightful discussions and fun. The support from my friends back home: Boris Jockov, Djukić Nenad, Miroslav Boljanović, Vladimir Kukić, and the others, was also invaluable. The beers we had together played an instrumental role in my completing this work.

Of course, I could have not become Dr. Aleksandar without all the support from my parents, Štefanija and Milenko, and my sister Tanja. All the insightful discussions we had (and will keep having) on Saturdays at 6pm helped immensely. Last but certainly not least, I would like to thank my wife Marija for all the selfless support, understanding, and love I have received from her. I feel truly blessed to have such a warm, supporting, and caring person to share my life with.

Lausanne, 14th May 2012 A. D.

Preface

The research leading to this thesis was conducted at the Distributed Programming Laboratory, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), under the supervision of Prof. Rachid Guerraoui in the period from 2007 to 2012.

The core of the thesis was published in the following papers:

Aleksandar Dragojević, Rachid Guerraoui, and Michał Kapałka. "Dividing Transactional Memories by Zero." 3rd ACM SIGPLAN Workshop on Transactional Computing (Transact 2008), 2008.

Aleksandar Dragojević, Rachid Guerraoui, and Michał Kapałka. "Stretching Transactional Memory." ACM SIGPLAN 2009 Conference on Programming Languages Design and Implementation (PLDI 2009), 2009.

Aleksandar Dragojević, Pascal Felber, Rachid Guerraoui, and Vincent Gramoli. "Why STM can be more than a Research Toy." Communications of the ACM (CACM), vol. 54, April 2011.

Besides work presented in the thesis, I carried out research resulting in several other publications, which I describe briefly in the chapter on related work:

Aleksandar Dragojević, Yang Ni, and Ali-Reza Adl-Tabatabai. "Optimizing Transactions for Captured Memory." 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA 2009), 2009.

Aleksandar Dragojević and Rachid Guerraoui. "Predicting the Scalability of an STM: A Pragmatic Approach." 5th ACM SIGPLAN Workshop on Transactional Computing (Transact 2010), 2010.

Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. "On The Power of Hardware Transactional Memory to Simplify Memory Management." 30th Annual ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing (PODC 2011), 2011.

Aleksandar Dragojević and Tim Harris. "STM in the Small: Trading Generality for Performance in Software Transactional Memory." EuroSys '12 the European Conference on Computer Systems (EuroSys 2012), 2012.

Aleksandar Dragojević and Rachid Guerraoui. "A Pragmatic Approach for Predicting the Scalability of Parallel Applications." Under submission (available as technical report EPFL-REPORT-174869).

I also helped with the research resulting in the following papers, but I was not the principal author:

Aleksandar Dragojević, Anmol Singh, Rachid Guerraoui, and Vasu Singh. "Preventing versus Curing: Avoiding Conflicts in Transactional Memories." 28th Annual ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing (PODC 2009), 2009.

João Baretto, Aleksandar Dragojević, Paulo Ferreira, Rachid Guerraoui, and Michał Kapałka. "Leveraging Parallel Nesting in Transactional Memory." 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010), 2010.

Abstract

The recent proliferation of multi-core processors has moved concurrent programming into mainstream by forcing increasingly more programmers to write parallel code. Using traditional concurrency techniques, such as locking, is notoriously difficult and has been considered the domain of a few experts for a long time. This discrepancy between the established techniques and typical programmer's skills raises a pressing need for new programming paradigms.

A particularly appealing concurrent programming paradigm is transactional memory: it enables programmers to write correct concurrent code in a simple manner, while promising scalable performance. Software implementations of transactional memory (STM) have attracted a lot of attention for their ability to support dynamic transactions of any size and execute on existing hardware. This is in contrast to hardware implementations that typically support only transactions of limited size and are not yet commercially available. Surprisingly, prior work has largely neglected software support for transactions of arbitrary size, despite them being an important target for STM. Consequently, existing STMs have not been optimized for large transactions, which results in poor performance of those STMs, and sometimes even program crashes, when dealing with large transactions.

In this thesis, I contribute to changing the current state of affairs by improving performance and scalability of STM, in particular with dynamic transactions of arbitrary size. I propose SwissTM, a novel STM design that efficiently supports large transactions, while not compromising on performance with smaller ones. SwissTM features: (1) mixed conflict detection, that detects write-write conflicts eagerly and read-write conflicts lazily, and (2) a two-phase contention manager, that imposes little overhead on small transactions and effectively manages conflicts between larger ones. SwissTM indeed achieves good performance across a range of workloads: it outperforms several state-of-the-art STMs on a representative large-scale benchmark by at least 55% with eight threads, while matching their performance or outperforming them across a wide range of smaller-scale benchmarks. I also present a detailed empirical analysis of the SwissTM design, individually evaluating each of the chosen design points and their impact on performance. This "dissection" of SwissTM is particularly valuable for STM designers as it helps them understand which parts of the design are well-suited to their own STMs, enabling them to reuse just those parts.

Furthermore, I address the question of whether STM can perform well enough to be practical by performing the most extensive comparison of performance of STM-based and sequential, non-thread-safe code to date. This comparison demonstrates the very fact that SwissTM indeed outperforms sequential code, often with just a handful of threads: with four threads

it outperforms sequential code in 80% of cases, by up to 4x. Furthermore, the performance scales well when increasing thread counts: with 64 threads it outperforms sequential code by up to 29x. These results suggest that STM is indeed a viable alternative for writing concurrent code today.

Keywords: Concurrent Programming, Software Transactional Memory, Performance, Scalability, Benchmarks, Conflict Detection, Contention Management

Résumé

La prolifération récente des processeurs multi-cœurs a rendu la programmation concurrente incontournable en forçant davantage de programmeurs à écrire du code parallèle. L'utilisation de techniques concurrentes traditionnelles, comme les verrous, est difficile et fut longtemps le domaine de peu d'experts. Ce décalage entre les techniques existantes et les compétences typiques des programmeurs illustre la nécessité de nouveaux paradigmes de programmation.

Un paradigme de programmation particulièrement attractif est la mémoire transactionnelle : elle permet à des programmeurs d'écrire simplement du code concurrent correct et offre des performances passant à l'échelle. Les implémentations logicielles de mémoire transactionnelle (MTLs) ont été au cœurs des attentions du fait de leur capacité à supporter des transactions dynamiques de toute taille sur le matériel existant. A l'inverse, les mémoires transactionnelles matérielles supportent typiquement des transactions de taille limitée et ne sont pas commercialisées. Il est surprenant d'observer que les travaux antérieurs ont négligé le support logiciel pour les transactions de taille arbitraire alors même qu'elles constituent une cible de choix pour les MTLs. Par conséquent, les MTLs ne furent pas optimisées pour de longues transactions, l'exécution de ces dernières induisant de mauvaises performances et provocant parfois même des erreurs non récupérables.

Dans cette thèse je contribue à modifier cet état de fait en améliorant les performances et le passage à l'échelle des MTLs, en particulier avec des transactions dynamiques de taille arbitraire. Je propose SwissTM, une nouvelle approche de MTL qui supporte efficacement les longues transactions sans entacher les performances des plus petites. SwissTM comprend : (1) une détection de conflit mixte qui résoud immédiatement les conflits écriture-écriture et plus tard les conflits lecture-écriture, et (2) un gestionnaire de contention en deux phases qui impose un faible surcoût sur les petites transactions et résoud efficacement les conflits entre les plus grandes. SwissTM fournit en effet de bonnes performances lors d'exécutions variées : il améliore les performances de quelques MTLs de référence sur une batterie représentative de tests à grande échelle par au moins 55% avec 8 fils d'exécution tandis qu'il présente des performances soit similaires soit meilleures sur des tests à plus petite échelle. Je présente également une analyse empirique détaillée du SwissTM, en évaluant individuellement chaque choix de conception et leur impact sur les performance. Cette "dissection" de SwissTM est particulièrement intéressante pour les programmeurs de MTLs pour mieux comprendre quelles sont les parties pouvant apporter des gains de performance significatifs dans à leur propres MTLs, les autorisant à réutiliser simplement ces parties.

Par ailleurs, je réponds à la question de la praticabilité des MTLs en exécutant la comparaison la plus détaillée à ce jour des performances des MTLs avec celles du code séquentiel. Cette comparaison montre précisément que SwissTM présente de meilleures performances que le code séquentiel, souvent avec seulement quelques fils d'exécution : avec 4 fils, les performances sont meilleures dans 80% des cas et jusqu'à 4x. De plus, les performances passent à l'échelle du nombre de fils : avec 64 fils, elles sont 29x meilleures que celles du code séquentiel. Ces résultats suggèrent que les STMs sont en effet une alternative viable pour écrire du code concurrent de nos jours.

Mots-clés: Programmation concurrente, Logiciel de Mémoire Transactionnelle, Performance, Evolutivité, Tests de performance, Détection de conflit, Gestionnaire de contention

Contents

List of Figures

List of Tables

1 Introduction

The improvements in sequential performance of processors have significantly slowed down in recent years due to increasing technical difficulties encountered by hardware manufacturers [112]. In response to the changed circumstances, the manufacturers have started building multi-core CPUs able to execute several threads in parallel, instead of sequential CPUs as before. The number of hardware threads supported by modern multi-core CPUs has kept increasing ever since, as well as the breadth of devices using them. As a consequence, today's CPUs support tens of simultaneous threads in hardware, and all mainstream devices, including desktops, laptops, tablets, and phones, have become multi-core [113]. This hardware revolution has had a profound effect on the way we think of and design *mainstream* software: to fully exploit the new multi-core CPUs, developers have to write concurrent programs. Whereas concurrent programming is not a novel discipline, it has long been reserved for computationally intensive problems, for which the investment in large, and expensive, parallel machines was justified. The multi-core revolution has made the parallel machines ubiquitous, thus moving concurrent programming from an obscure discipline practiced by only a handful of experts into the mainstream, where everyone has to do it.

1.1 Traditional concurrent programming

Traditional paradigms for concurrent programming, such as locking and non-blocking techniques, are notoriously difficult. This makes them unsuitable for use by average programmers who write the mainstream applications. The following example illustrates some of the drawbacks of using locking, which is the best known and most widely used concurrent programming paradigm. The example considers several bank accounts, as illustrated in Figure 1.1. Each of the accounts has an associated identifier (acc_1–acc_5) and balance (20 on acc_1, 50 on acc_2, etc.). Let us consider a problem of implementing a transfer of a specified amount from one bank account to another. The following is a typical requirement for such an operation: during the execution of the transfer operation, the total amount on all bank accounts observed by other threads has to remain constant. In particular, the other threads are not allowed to observe the new balance on one of the accounts and the old one on the other. This correctness

Figure 1.1: Bank accounts example.

```
1 void transfer(account_t from, account_t to, int amount) {
2     from.acquire();
3     to.acquire();
4
5     from.withdraw(amount);
6     to.deposit(amount);
7
8     to.release();
9     from.release();
10 }
```

Figure 1.2: A fine-grained implementation of transfer operation, prone to deadlocks.

criterion is called *linearizability* [65] and is typically ensured by concurrent data structures and algorithms.

To eliminate inconsistencies in the executions and ensure linearizability, programmers may use locks. Locks are synchronization objects that threads *acquire* before accessing shared data. At any given point in time, a lock can be acquired, or owned, by a single thread. If a thread attempts to acquire a lock while it is being held by another thread, it will be blocked until the owner of the lock *releases* it. The simplest approach to correctly implementing the transfer operation with locks is to use a single lock, that is acquired by threads before they perform any operation involving the bank accounts. This is the simplest possible locking protocol: it serializes all accesses to the shared data and is, thus, obviously correct. The major drawback of this approach, however, is that it does not allow multiple independent operations to be performed concurrently. For example, a transfer from acc_1 to acc_2 is completely independent from a transfer from acc_3 to acc_4, yet the solution based on a single lock does not permit their parallel execution.

To allow more parallelism and, thus, improve performance of the transfer operation, a finer-grained locking scheme has to be used. Associating a different lock to each of the accounts enables threads to independently lock them, allowing execution of independent transfer operations in parallel. The pseudo-code of a transfer operation that uses a lock per account is given in Figure 1.2. It enables execution of the two transfer operations from above in parallel, as they do not access any accounts in common. However, the code in Figure 1.2 suffers from a serious problem, as illustrated in Figure 1.3. Figure 1.3 represents the bank accounts as squares and the threads as ovals, with solid arrows representing locks that have been acquired by threads and dashed arrows representing threads waiting for locks. In the figure, two transfer operations are executed in parallel. Thread A performs a transfer from acc_2 to acc_4 and thread B performs a transfer from acc_4 to acc_2. Each of the threads has successfully acquired the lock of its origin account, and is waiting for the lock of the destination account to be released. In

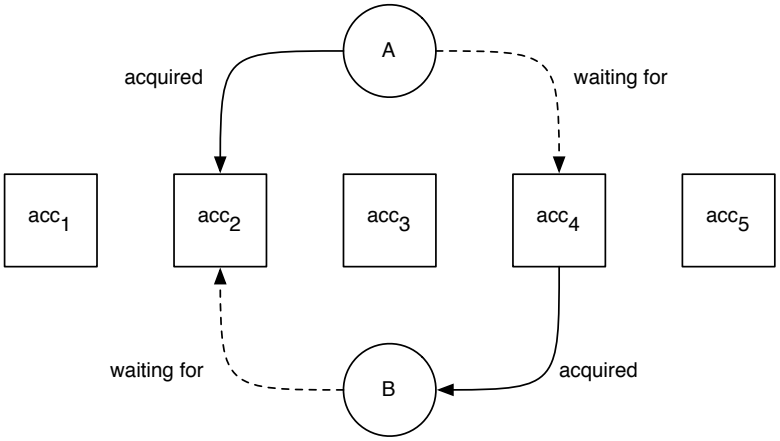


Figure 1.3: A deadlock in the bank accounts example due to incorrect use of locking.

this execution, neither of the threads is able to make progress, as each is waiting for the other to release a lock. This is an example of a well known locking bug, called *deadlock*. A simple approach to avoiding deadlocks is to always acquire the locks in the same order, independently of the order in which the objects are accessed. This solution can be applied to our example as well, as illustrated in Figure 1.4.

As the pseudo-code shows, the solution to the deadlock problem is rather straightforward in this simple example. However, it is often very difficult to *detect* that the threads can indeed deadlock, even for experienced programmers. The main reason is that the deadlocks, as well as other concurrency bugs, are not necessarily manifested in every execution of the program. This means that they might remain hidden even after extensive testing. For example, the incorrect transfer operation in Figure 1.2 does not cause a deadlock if, during testing, the identifier of the origin account is always lower than that of the destination account, or vice versa. Even if this is not the case, the bug may be masked by a "lucky" scheduling of critical operations. For example, it is possible that one of the threads is always quicker to acquire both required locks, in which case the deadlock does not manifest itself during testing. This non-determinism in executions of parallel programs makes it extremely difficult to reason about them and ensure that they are correct.

Another significant drawback of locking is the lack of composability. As an example, consider a new operation that transfers some amount of money from two accounts to a destination account. As in the previous example, the operation has to be linearizable, meaning that the intermediate results of the operation must not be visible to other threads. In particular, this means that it is not allowed to have other threads observe the effects of only one of the transfers. It is clear that the correct implementation of a single account transfer does not help with implementation of this new two-account transfer, as simply invoking two single account transfers one after the other allows other threads to observe partial transfers, thus violating linearizability. Instead, the new operation has to be implemented from scratch, and it has to respect the locking protocols used by existing operations, such as to acquire the locks in a

```
1 void transfer(account_t from, account_t to, int amount) {
2     if(from.id < to.id) {
3         from.acquire();
4         to.acquire();
5     } else {
6         to.acquire();
7         from.acquire();
8     }
9
10    from.withdraw(amount);
11    to.deposit(amount);
12
13    to.release();
14    from.release();
15 }
```

Figure 1.4: A correct fine-grained implementation of the transfer operation.

particular order, or otherwise risk introducing hard-to-detect bugs. The lack of composability of locking makes it hard to extend and maintain existing lock-based programs, especially if they are written and maintained by different programmers.

There exist several other well known problems with locking, such as priority inversion and convoying, but discussing them in detail is outside of the scope of this thesis. For more details on these issues, see [103].

Non-blocking[1] programming solves most of the issues with locking, but is even more difficult to write correctly. Non-blocking algorithms for relatively simple data structures, such as rendezvous object [6], task pool [13], and bag collection class [111], are still publishable results in top conferences on parallel computing. This means that even the concurrency experts cannot be expected to write complex non-blocking algorithms on a daily basis.

To make concurrent programming truly widespread, as required by the changing hardware landscape, it has to be made simple. It is clear that the traditional programming paradigms are not simple enough to be used by average programmers, and, therefore, new paradigms are needed. Ideally, concurrent programming should be as simple as when using a single lock, to make it accessible to average programmers, but the performance and scalability of resulting code need to be better, if possible as good as when using fine-grained locking. A new concurrent programming paradigm, *transactional memory* (TM) [64], promises precisely that.

1.2 Transactional memory

With transactional memory, programmers simply mark sections of sequential code that need to be executed atomically with respect to other threads. The underlying TM system ensures that these atomic blocks of code, also called *transactions*, are (1) executed indivisibly, meaning

[1] In this thesis all algorithms that do not rely on locks are referred to as non-blocking. This includes wait-free, lock-free, and obstruction-free algorithms.

```
1  void transfer(account_t from, account_t to, int amount) {
2      atomic {
3          from.withdraw(amount);
4          to.deposit(amount);
5      }
6  }
7
8  void transfer_2(account_t from_1, account_t from_2, account_t to, int amount) {
9      atomic {
10         transfer(from_1, to, amount);
11         transfer(from_2, to, amount);
12     }
13 }
```

Figure 1.5: A TM-based implementations of the transfer operations.

that either all or none of the statements from transactions get executed, and (2) in isolation one from another, meaning that transactions do not observe partial results of other transactions. The TM paradigm is particularly appealing as it greatly simplifies concurrent programming compared to the traditional techniques, while still promising good performance: programmers only need to reason about which code has to be executed atomically and the TM ensures its atomic execution and good performance.

To illustrate the use of TM, the same transfer operations as above are implemented using transactions in Figure 1.5. The code of the transfer operation is as simple as the code that uses a single lock, and it fully relieves the programmer of worrying about low-level synchronization details. Due to the simplicity of the TM paradigm, the code is also obviously correct. Furthermore, it composes well: as the figure shows, the two-account transfer can be simply implemented by composing two single-account transfers.

Transactional memory has attracted much attention lately because of its great potential. Many hardware (HTM) [4, 20, 21, 27, 53, 64, 83, 89, 91, 116], software (STM) [3, 16, 24, 28–30, 41, 43, 56, 57, 63, 69, 76, 77, 85, 99, 101, 105, 109, 115], and hybrid hardware-software (HyTM) [22, 25, 70, 81, 95] implementations have been proposed. I do not discuss these proposals in detail here. Instead, some of them are discussed in Chapter 6, and for a more comprehensive TM survey, see [55].

Hardware implementations have several appealing characteristics including high performance, clean semantics, and seamless integration with legacy code. However, their deployment costs are significant: implementing HTM requires making changes to the CPUs, which hardware manufacturers are reluctant to do. As a result, actual HTM proposals from the industry have started to appear only relatively recently [4, 27, 91], and none of them are commercially available yet. These HTMs typically support only best-effort execution of limited-size transactions: transactions can access only a limited number of memory locations, for example RockTM [27] transactions can update at most 32 locations inside a single transaction, and they may fail spuriously for implementation-specific reasons, for example when cache misses or interrupts occur during transaction execution. The bounded nature of HTMs is their biggest

limitation, apart from the lack of actual implementations, as it prevents HTMs from being used in a wide range of programs. In contrast, STM proposals support dynamic, unbounded transactions and can be implemented as user-level libraries, which makes their deployment simple and easy. STM does have its disadvantages too: it has lower performance than HTM and its semantics are not always as clean. HyTMs try to take the best of both worlds: they execute transactions supported by the HTM in hardware, which are typically short and simple transactions, and the remaining transactions in software. In this way the short and simple transactions benefit from the existing HTM, whereas the TM still supports all transactions required by programs.

In this thesis, I focus on STM, precisely because it supports dynamic, unbounded transactions and because it does not require any modifications to the underlying hardware.

1.3 Software transactional memory

An important target for TMs are large applications such as business software or video games: the size of these applications makes them ideal candidates to benefit from parallelization and emerging multi-core architectures. Such applications typically involve dynamic and non-uniform data structures consisting of many objects of various complexity. For example, a video gameplay simulation can use up to 10,000 active interacting game objects. Each of the objects has mutable state and is being updated 30–60 times per second, where every update causes changes to 5–10 other objects [114]. Unless a TM is used, making such code thread-safe and scalable on multi-cores is a daunting task [114]. The big size and complexity of such applications can, in turn, easily lead to use of large transactions, for these can naturally be composed [56]. Some TM interfaces, in fact, promote the encapsulation of entire applications within very few transactions [1].

Executing such large transactions in hardware is not possible with current HTM proposals. This means that STM will keep being of practical relevance even after the hardware TM support becomes widely available in the future, as it is likely that only smaller-scale transactional workloads will be fully executed in hardware, while software support will still be needed for larger-scale transactions. For example, HyTM proposed in [70] switches from full hardware TM to full software TM when it encounters large transactions, and other HyTMs use similar techniques. Consequently, the ability of STM systems to effectively deal with large transactions will be crucial in these settings.

The main reason for writing concurrent code is to exploit the parallelism exposed by the hardware and, thus, improve the performance of the code. If performance of programs written using STM is poor, it is highly unlikely that the developers will decide to use STM. Instead, they will wait for the upcoming hardware support, and keep using complex, but fast locking techniques until HTM becomes available. Furthermore, it is likely that programs with naturally large operations that cannot be executed by HTM will keep being written using

locking techniques, even after HTM is widely spread. This makes the performance of STM a crucial concern, especially for workloads that consist of large and complex transactions.

In this thesis, I address two important questions regarding the performance of STM:

1. How to design and implement an STM that performs particularly well with workloads consisting of large and complex transactions, while still having good performance with other kinds of workloads?

2. Can STM performance be appealing for practical deployment scenarios? More precisely, can STM outperform sequential, single-threaded code, and if so, with how many threads?

1.4 Contributions

Since the seminal paper on STM that supported dynamic data structures and unbounded transactions [63], all modern STMs are supposed to handle complex workloads [3, 24, 28, 29, 43, 57, 63, 77, 85, 92, 99]. A wide variety of STM techniques, mainly inspired by database algorithms, have been explored. The big challenge facing STM researchers is to determine the right combination of strategies that suit the requirements of concurrent applications, which are significantly different than those of database applications. So far, however, most STMs have been evaluated using benchmarks characterized by small transactions, simple and uniform data structures, or regular data access patterns. While such experiments reveal some of the performance differences between STM implementations, they are not fully representative of complex workloads that STMs are likely to get exposed to when used in real applications. Worse, they can mislead STM designers by promoting certain strategies that may perform well in small-scale applications but are counter-productive with complex workloads.

To change this state of affairs, I have implemented STMBench7 [52], a large-scale benchmark for STMs, with several state-of-the-art STMs. The implementation and experimentation with STMBench7 lead to several surprising conclusions. First, performance results I gathered differ from previously published results. I found, for instance, that conflict detection and contention management have the biggest performance impact with large transactions, significantly more than other aspects, like the choice of lock-based or obstruction-free implementation, as was typically highlighted. Next, most STMs I used crashed, at some point or another, when running STMBench7, mainly due to memory management limitations. This means that, in practice, none of the used STMs were truly unbounded and dynamic, which are the main motives for moving away from HTM. Whereas the discovered bugs were usually easy to fix, the fact that these STMs crashed illustrates that they were not thoroughly tested with large-scale workloads, meaning that they were not optimized for them either. Finally, implementing STMBench7 with various STMs also revealed several programming related issues such as the lack of support for external libraries and partial support for object oriented language features. These issues are likely to be a major limitation when adapting STMs for production use.

Using the results and experiences from the experiments with STMBench7, I revisit the main STM design choices from the perspective of complex workloads and propose a new STM, called SwissTM. In short, SwissTM is lock- and word-based STM that uses (1) pessimistic, or encounter-time, conflict detection for write-write conflicts and optimistic, or commit-time, conflict detection for read-write conflicts and (2) a new Two-phase contention manager that efficiently deals with conflicts among long transactions while inducing no overhead on short ones. SwissTM outperforms state-of-the-art STM implementations, namely RSTM [77], TL2 [28], and TinySTM [43], in the experiments on STMBench7 [52], STAMP [17], Lee-TM [8], and the red-black tree benchmark, demonstrating good performance on a wide range of workloads, not just the large-scale ones. Beyond SwissTM, I present the most complete evaluation to date of the individual impact of various STM design choices on the ability to support the mixed workloads of large applications.

Next, I use SwissTM to answer an important question of whether the STM performs well enough to actually be used in practice, which was, surprisingly, largely neglected in the previous work. The most notable previous study that addressed this question [19], concluded that the overheads of using STM are too high for it to be of practical relevance. In the experiments from [19], based on several micro benchmarks and a subset of STAMP benchmark suite, even with eight threads STM did not perform as well as sequential, single-threaded code, and for that it was called a "research toy". I revisit these conclusions through the most extensive comparison of STM performance to sequential code to date, using a wide range of benchmarks and two different multi-core systems. The goal of the evaluation is to understand whether the STM-based code can outperform the sequential code, and if it can, how many hardware threads it requires to do so. This question is important as, after all, writing parallel code using STM requires only slightly more effort than writing the equivalent sequential code, and if its performance is better, the programmers might decide to use STM even though it does not match the performance of the more complex synchronization techniques.

This study shows the performance of STM to be much better than previously claimed, as SwissTM outperforms the sequential code in most cases. In the evaluation presented in [19], STM outperforms sequential code by at most 2.5x on systems with eight hardware threads, and, in most cases, it fails to outperform it at all. In contrast, my evaluation shows that SwissTM outperforms sequential code by more than 9x in the best case on an x86 system with 16 hardware threads, and breaks even or outperforms sequential code using 4 threads in 13 out of 17 cases. Similar results were obtained on a SPARC system with 64 hardware threads where SwissTM outperforms equivalent sequential code by more than 29x in the best case and breaks even or outperforms it using 4 threads in 14 out of 17 cases. The evaluation also shows that, while the overheads of compiler instrumentation and transparent privatization are substantial, they do not prevent STM from generally outperforming sequential code.

These performance results demonstrate that STM can do well across a wide range of workloads and multi-core architectures. Whereas I do not claim that STM is a silver bullet for general purpose concurrent programming, the presented results contradict the previous experiments [19]

and suggest that STM is already now a viable option for various types of applications. The results support the initial hopes about STM performance and motivate further research in the field.

To summarize, the contributions of this thesis are:

1. An evaluation of the ability of several state-of-the-art STMs to correctly and efficiently handle workloads with large transactions, which reveals several surprising conclusions about performance of STMs. It also shows that none of the used STMs have been thoroughly tested with large-scale, complex workloads.

2. The design and implementation of SwissTM, an STM that performs particularly well with large-scale complex transactional workloads while having good performance with smaller-scale ones.

3. An extensive experimental evaluation of STM design and implementation techniques from the perspective of complex applications with mixed workloads.

4. The most extensive comparison of STM performance to sequential code to date, using a wide range of benchmarks and two different multi-core systems, which demonstrates that STM could already now be used in practice for certain types of applications.

5. An experimental evaluation of the inherent costs of STM synchronization, as well as the overheads of compiler instrumentation and transparent privatization. This evaluation shows that STM can achieve good performance and expose a clean programming model when using an STM compiler, and requiring programmers to explicitly annotate transactions that privatize data.

At the time of this writing, the code of SwissTM and benchmarks resulting from work on this thesis is available for download from Transactions@EPFL [71].

1.5 Outline

The rest of the thesis is organized as follows.

Background on software transactional memory is given in Chapter 2. This includes a brief description of transactions and general phases in their execution (Section 2.1); a discussion of correctness criterion called *opacity* [51] typically guaranteed by TM implementations (Section 2.2); a brief description of STM programming interface and its usage (Section 2.3); and a summary of several semantic limitations of STM (Section 2.4).

The results and conclusions from experiments with STMBench7 [52], a large-scale STM benchmark, and several state-of-the-art STMs are discussed in Chapter 3. This includes an overview of the STM design space (Section 3.2); a description of STMBench7 and its implementation

with word-based STMs in C++ (Section 3.3); the main conclusions regarding performance of STMs with large transactions (Section 3.4); a summary of several surprising bugs in state-of-the-art STMs that were uncovered by the use of STMBench7's large transactions (Section 3.5); and some findings on programmability limitations of the used STMs (Section 3.6).

The design, implementation, and experimental evaluation of SwissTM, a new STM inspired by the conclusions from the previous chapter is presented in Chapter 4. I describe SwissTM's design and implementation (Section 4.2); compare its performance to other state-of-the-art STMs on a number of benchmarks with varying characteristics (Section 4.3); evaluate in detail the impact of the design and implementation choices on the performance (Section 4.4); and describe how to integrate SwissTM with standard STM compilers and extend it with support for the privatization idiom (Section 4.5).

An extensive comparison of SwissTM's performance to performance of sequential code is presented in Chapter 5. This includes a summary of overheads incurred by an STM (Section 5.1); a performance evaluation of four different SwissTM configurations each with different programming model and performance characteristics (Sections 5.3–5.7); and the conclusions about the most appealing STM configuration, which exposes reasonably simple programming model and has good performance (Section 5.8).

In Chapter 6, I present the related work, which also covers the work I did in parallel with this thesis. I summarize the thesis and outline several exciting opportunities for future work in Chapter 7.

2 Background

In this chapter, I provide general background on TM, focusing in particular on STM. I first give an overview of transactional execution, discussing the notion of *conflicts* between transactions and transaction *commit* and *abort*. Then, I briefly describe *opacity* [51], a correctness criterion most often used by TM implementations, followed by introduction of the typical word-based and object-based STM interfaces used throughout the thesis. I also discusses some alternatives to opacity and several semantical limitations of STMs, such as *weak atomicity* [79] and *publication* and *privatization* safety [106].

2.1 Transactional execution

The previous chapter introduces TM as a paradigm that promises programming as simple as with a single lock, with performance of resulting programs that is as good as with the fine-grained locking. It also briefly illustrates why the programming is indeed simple: programmers only need to mark the atomic sections, similarly to acquiring the single lock, and the underlying TM takes care of the rest. Recent user studies largely confirm that programming with TM is indeed simpler than with the alternative synchronization techniques, even when TM lacks certain functionality and adequate tools [87, 97] Next, I discuss techniques typically used by TMs for achieving performance of fine-grained locking and outline the common steps of transactional execution.

To achieve good performance, most TM implementations, both in hardware and in software, adopt a similar approach to the fine-grained locking examples from the previous chapter. More precisely, they track transactional accesses at a fine granularity, typically using granularity of a memory word, a cache line, or an object, and detect when accesses performed by concurrent transactions could lead to incorrect executions, such as executions in which a transaction is allowed to observe intermediate results of another transaction. To prevent such incorrect executions, TM *aborts* one of the transactions, for example the one about to observe partial results of another transaction. At that point, all actions performed by the aborted transaction are reverted, effectively voiding the whole transaction. This is commonly referred

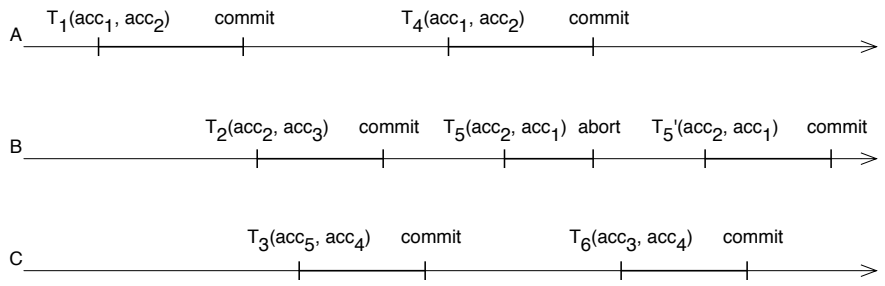


Figure 2.1: An example execution of transactional transfer operations between bank accounts.

to as transaction *rollback*. Typically, the aborted transaction is restarted in hope that it will successfully complete when it is next attempted. When a transaction successfully completes, it *commits* all of it changes to memory. At that point, all of the changes it performed become visible to other threads atomically.

Figure 2.1 depicts an execution of several transactional transfer operations from the previous chapter. Three threads, A, B, and C, execute transactions, with time flowing from left to right. Each transaction performs a transfer between two accounts, as indicated in the brackets. Transaction T_1 executes alone, with no transaction concurrent to it and, therefore, successfully commits. Transactions T_2 and T_3 are concurrent, but they access different accounts, so they also commit. Transactions T_4 and T_5, however, concurrently access the same accounts acc_1 and acc_2. To prevent inconsistencies in the execution, TM aborts transaction T_5, allowing transaction T_4 to commit. Transaction T_5 is restarted and re-executed as transaction T_5'. T_5' successfully commits as there are no concurrent transactions that access the same bank accounts: transaction T_6 is concurrent with T_5', but it accesses accounts acc_3 and acc_4.

It is important to note that, to facilitate aborts, transactions log their updates. They either log the old values of updated objects, using so called *in-place updates* and *undo logging*, or the new values to be written, using *deferred updates* and *redo logging*. With in-place updates, transactions directly update the accessed objects, logging their old values before the update. The logged values are used during the rollback to restore the objects to their old values if the transaction aborts. With deferred updates, transactions do not update objects during their execution, instead storing the new values into the log. The logged values are used to update the objects when the transaction commits.

One simple approach to implementing a TM is to associate a lock to each shared object and have every transaction acquire the lock corresponding to the object before accessing it. Transactions release the locks of all accessed object when they commit or abort. With such an implementation, the locks serve to track transactional accesses to shared objects. If two transactions try to access the same object concurrently, we say that these accesses *conflict* with each other. In the case when two accesses conflict, one of the transactions performing them will get aborted to prevent inconsistencies in the execution. The main difference between such an implementation of TM and fine-grained locking is that the TM

logs the updates, which allows it to abort transactions and avoid the deadlocks that naive fine-grained locking implementations suffer from. In practice, TM usually aborts transactions even before a deadlock occurs, typically as soon as it detects a conflict among concurrent transactions, to simplify the way conflicts are detected.

To enable more parallelism, TM implementations usually distinguish between read and write accesses to objects and allow several transactions to read the same object concurrently, as long as no transaction is updating the object at the same time. This is safe because inconsistencies cannot arise when transactions are only reading the object. When a TM distinguishes between read and write accesses, the notion of conflicting accesses changes: two transactions conflict when they access the same object and when at least one of the accesses is a write.

Whereas I introduced the transactional execution and terminology using the analogy with fine-grained locking, TM implementations do not necessarily have to rely on locking. For example, most HTM implementations detect the conflicting accesses using the existing cache coherence mechanisms [27, 53, 64, 83, 89]. Similarly, many STMs either do not use traditional locks at all [46, 63, 77, 109, 115], or use locks only when performing writes, relying on optimistic techniques, such as read validations, to ensure consistency of their reads [24, 28, 43, 77, 92, 99]. These techniques and the design space of STM is discussed in more detail in Section 3.2.

2.2 Opacity

While reasoning about conflicting transactional accesses can be useful for intuitively understanding TM correctness, it is not always precise enough. Therefore, a formal correctness criterion for TM is needed. Next, I describe *opacity* [51], the most widely used formal correctness criterion for TM. Opacity precisely specifies when a TM is correct and enables reasoning about correctness in a formal way.

Opacity builds on a well known database correctness criterion, called *serializability* [88]: an execution of several transactions is serializable if there exists an equivalent execution of all committed transactions by a single thread, which is also called a sequential execution. For a TM to be serializable, it has to allow only serializable executions. To serializability, opacity adds a requirement that transactions have to observe consistent states of shared objects at all times. Alternatively stated, opacity requires that the equivalent sequential execution includes the aborted transactions in addition to the committed ones, where the aborted transactions have no effect on the shared objects.

The main reason for including the additional requirement to serializability is to eliminate execution exceptions, such as divisions by zero or accesses to deallocated memory, that can occur in parallel executions if transactions that observe an inconsistent state of shared objects, and are thus doomed to abort, are allowed to keep executing. Such exceptions break the illusion that the execution is equivalent to a sequential execution: they cannot occur

```
1  // Invariant: x < y
2  int x = 0, y = 1, z;
3
4  void OpA() {
5    atomic {  // T1
6      int xloc = x;
7      int yloc = y;
8      x = yloc;
9      y = yloc * 2;
10   }
11 }
```

```
12 void OpB() {
13   atomic { // T2
14     int xloc = x;
15     int yloc = y;
16     // no need to check ylock != xlock
17     // due to the invariant
18     int zloc = 1 / (yloc - xloc);
19     z = zloc;
20   }
21 }
```

Figure 2.2: Code that can cause division by zero exceptions if transactions do not observe consistent object states.

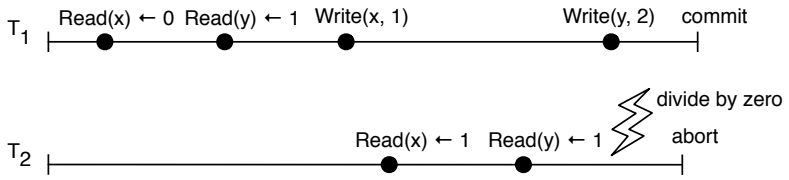


Figure 2.3: Example execution of pseudo-code from Figure 2.2 in which a divide by zero exception occurs if consistency of reads is not guaranteed.

in any sequential execution as transactions always observe consistent state when executed sequentially.

Figure 2.2 presents the pseudo-code of simple operations that suffer from the described problem. The pseudo-code defines two operations, `OpA` and `OpB`, that use shared integer variables `x`, `y`, and `z`. Both operations leave the system in a state where `x` < `y`, so the division at line 18 is safe if the operations are executed by a single thread. However, if the operations are executed by two concurrent threads which are allowed to observe inconsistent states of the variables, a divide by zero exception can occur, as illustrated in Figure 2.3. In the figure, transaction T_1 is executed to line 8, setting `x` to the value of `y` and temporarily violating the invariant. With serializability, transaction T_2 is allowed to observe the inconsistent state of the variables, where `x` and `y` are equal, and execute up to the point of commit, as long as it is aborted at commit time. However, the division at line 18 will cause an exception, resulting in behavior that is not possible in any sequential execution.

Division by zero is not the only problem that can occur if transactions are allowed to execute on inconsistent memory states. Other examples of possible problems include accesses to deallocated memory and infinite loops [51, 77]. Opacity prevents such problems as it requires transactions to abort as soon as they encounter the inconsistent state, without returning the inconsistent value to the user code. In our example, transaction T_2 would have to abort at line 15 if it observed the new value of `x` and the old value of `y`, avoiding the described problem.

I omit the formal definition of opacity, as it is out of the scope of this thesis. Interested readers can find more details in [51]. I briefly discuss alternatives to opacity in Section 2.4.

```
1 // architecture-specific word type
2 typedef word_t;
3 // system-defined long jump buffer
4 typedef jmpbuf_t;
5 // transaction-local descriptor
6 struct TxDescriptor;
7
8 // starts a new transaction
9 void TxStart(TxDescriptor *desc);
10 // commits the current transaction
11 void TxCommit(TxDescriptor *desc);
12 // from transaction, read a word at the specified address
13 word_t TxReadWord(TxDescriptor *desc, word_t *address);
14 // from transaction, write a new value to the word at the specified address
15 void TxReadWord(TxDescriptor *desc, word_t *address, word_t value);
```

Figure 2.4: Typical word-based STM interface.

To summarize, opacity intuitively requires that:

1. The effects of committed transactions become visible to other transactions at a single, indivisible moment during the transaction execution.

2. The effects of aborted transactions are never visible to other transactions.

3. All transactions, including the aborted ones, observe a consistent state of the system.

2.3 STM interface

So far, the pseudo-code examples used the `atomic` keyword to denote transactions, which is the simplest and the cleanest interface that a TM can expose to the programmers. Next, I discuss two alternative lower-level interfaces, which are used in the rest of the thesis: a *word-based* and an *object-based* interface. The interface calls can be directly inserted into code by programmers when writing transactional programs, or, alternatively, can be inserted by an STM compiler during compilation of atomic code blocks.

Word-based. A typical word-based STM interface is given in Figure 2.4. The figure uses a C/C++-like syntax to make a clear distinction between values and addresses of memory locations. It supports read and write accesses at the architecture-specific word granularity. Each thread initializes and uses a thread-local transactional descriptor object, where it stores all the information required for transactional book-keeping. The descriptor is passed as a parameter to all interface calls, to enable them to maintain the transaction state during the execution. The basic interface consists of four calls: for starting and committing transactions and for reading and writing memory words from inside transactions. If a transaction gets aborted, it is restarted using a system-provided long jump mechanism, which is used to transfer the control to the start of the transaction.

```
1 void transfer(account_t *from, account_t *to, int amount) {
2     // get the correctly initialized descriptor for this thread
3     TxDescriptor *desc = GetDescriptor();
4     TxStart(desc);
5     int from_amount = TxReadWord(desc, &from->amount);
6     int to_amount = TxReadWord(desc, &to->amount);
7     TxWriteWord(desc, &from->amount, from_amount - amount);
8     TxWriteWord(desc, &to->amount, to_amount + amount);
9     TxCommit(desc);
10 }
```

Figure 2.5: Transfer operation implemented using the word-based STM interface.

The familiar bank account transfer operation implemented using the word-based STM interface is given in Figure 2.5. The example conveys that the direct use of the word-based interface can be tedious and error prone, as it requires the programmer to correctly insert STM calls for each transactional access to shared data. If any of the accesses is, mistakenly, performed using an ordinary CPU instruction instead, the program will not work correctly and will exhibit hard-to-detect bugs, typical for parallel programs. This defeats the main purpose of TM, as such bugs are exactly what TM was devised to eliminate. To avoid such problems and truly deliver on the promise of easy programming, an STM compiler that enables the use of source-level atomic blocks is needed [23]. To translates the atomic blocks, the compiler inserts calls to the word-based interface, generating similar code to the one in the figure.

Starting from the presented basic word-based interface, it is straightforward to implement a more complete one that supports accesses to data types of different size, such as bytes, floats and doubles, which are necessary for general-purpose programming. I describe how to do so in more detail in Section 4.5.1.

In the remainder of the thesis, I use the STM interface directly, unless stated otherwise. Doing so enables me to evaluate the impacts of the STM design and implementation on the performance without the overheads that the compiler might introduce [38, 121]. These overheads are discussed in more detail in Sections 4.5.1 and 5.5.

Object-based. An example of an object-based STM interface is given in Figure 2.6. The interface supports read and update accesses at the granularity of transactional objects. This means that each user class that needs to be accessed from transactions has to be inherited from an STM-defined base transactional class, named `TxObject` in the figure. Similarly to the word-based STM, each thread initializes and uses a thread-local transactional descriptor object, where it stores all the information required for transactional book-keeping and which is passed as a parameter to all interface calls. The example interface consists of four calls: for starting and committing a transaction and for opening transactional objects for read and update accesses. An object has to be opened for reading before it is read and for updating before it is updated. Language constructs can sometimes be used to enforce these rules and help prevent bugs resulting from missing or inadequate open calls. In C/C++, for example, if

```
1 // base transactional type
2 class TxObject;
3 // transaction-local descriptor
4 class TxDescriptor;
5
6 // starts a new transaction
7 void TxStart(TxDescriptor desc);
8 // commits the current transaction
9 void TxCommit(TxDescriptor desc);
10 // from transaction, prepare the object for reading
11 void TxOpenForReading(TxDescriptor desc, TxObject object);
12 // from transaction, prepare the object for updating
13 void TxOpenForWriting(TxDescriptor desc, TxObject object);
```

Figure 2.6: Typical object-based STM interface.

```
1 void transfer(account_t from, account_t to, int amount) {
2     // get the correctly initialized descriptor for this thread
3     TxDescriptor desc = GetDescriptor();
4     TxStart(desc);
5     TxOpenForWriting(desc, from);
6     TxOpenForWriting(desc, to);
7     int from_amount = from.GetAmount();
8     int to_amount = to.GetAmount();
9     from.SetAmount(from_amount - amount);
10    to.SetAmount(to_amount + amount);
11    TxCommit(desc);
12 }
```

Figure 2.7: Transfer operation implemented using the object-based STM interface.

the client gets a pointer to a constant object when opening the object for reading, it cannot update the object afterwards. Aborted transactions can be restarted using the long jump mechanism, similarly to the word-based STMs, or, alternatively, they can be restarted by throwing an STM-specific exception type that is caught outside of transactions.

An implementation of the familiar bank account transfer operation using the object-based STM interface is given in Figure 2.7. Programming is slightly less tedious and error prone than with the word-based interface as the objects need to be opened in the correct mode of access only once, and can then be accessed several times. Therefore, there are fewer opportunities for programmers to omit the necessary STM calls. For example, the pseudo-code that uses an object-based STM contains only two calls for opening objects inside the transaction,[1] whereas the pseudo-code that uses a word-based STM contains four STM calls. Still the mistakes can occur easily, and forgetting to open an object in the correct mode of access results in hard-to-detect bugs, as discussed above.

Word-based STMs have become predominant for lower-level languages such as C/C++ mainly because they are more general and easier to integrate with STM compilers. In this thesis, I focus on C/C++ as it enables me to understand the performance of STM without reasoning about the

[1] In the example I assume that it is allowed to read the object opened for update, which is typically supported by TMs.

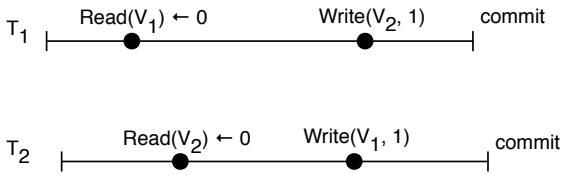


Figure 2.8: Transaction execution that is permitted by snapshot isolation, but not by opacity.

artifacts of higher-level languages, such as Java or C#, and their runtime systems. Because of that I mostly focus on the word-based STMs. Object-based STMs do have certain advantages when implementing complex, inherently object-based systems. Chapter 3 describes how an object-based interface can be built on top of a word-based one when necessary.

2.4 STM semantics

Next, I describe two alternatives to opacity, which have been adopted by several STM systems. Also, I discuss the difference between *strong atomicity*, typically ensured by HTM systems, and *weak atomicity*, which is typically ensured by STMs.

Alternatives to opacity. Whereas opacity is a widely used correctness criterion for TMs, there are several alternatives that have also been adopted, mostly by STMs. Here I mention two: *snapshot isolation* [14], adopted by, for example, LSA-STM [92], and serializability with *sandboxing*, adopted by, for example, JudoSTM [86].

With snapshot isolation, similarly to opacity, transactions observe a consistent snapshot of memory and they commit all their values atomically. However, transactions are allowed to commit the values at a later point than the point of the snapshot, as long as the objects they are updating have not changed since the snapshot was taken. Alternatively stated, each transaction corresponds to two indivisible points in the equivalent sequential execution: all reads happen at one point and all writes happen at the other, later point in time. In contrast, with opacity and serializability, all the reads and writes occur at the same point. Snapshot isolation eliminates the problems caused by transactions operating on inconsistent states of memory, such as the division by zero example in Section 2.2. However, it provides a less intuitive model for reasoning about transactions. With snapshot isolation transactions do not behave as if they acquired a single global lock when starting and released it when committing, as is the case with opacity. Instead, transactions roughly behave as if they acquired and released the lock twice: once for reading and once, later, for writing.

Figure 2.8 depicts an execution that is allowed by snapshot isolation but is not allowed by opacity and serializability. In the figure, transaction T_1 reads value 0 from variable V_1 and writes value 1 to V_2. Concurrently to T_1, transaction T_2 reads value 0 from V_2 and writes 1 to V_1. With snapshot isolation, both transactions are allowed to commit, as they operate on

```
1  // Invariant: x < y
2  int x = 0, y = 1, z;
3
4  void OpA() {
5    atomic {  // T1
6      int xloc = x;
7      int yloc = y;
8      x = yloc;
9      y = yloc * 2;
10   }
11 }
```

```
12 void OpB() {
13   atomic { // T2
14     int xloc = x + 1;
15     int yloc = y;
16     while(xloc != yloc) {
17       do_something(xloc);
18       xloc++;
19     }
20   }
21 }
```

Figure 2.9: Code that can loop indefinitely if transactions' reads are not consistent.

consistent snapshots and the data they are updating has not changed since the snapshot was taken. With opacity, however, one of the transactions would have to abort, as no equivalent sequential execution of transactions T_1 and T_2 exists. Snapshot isolation is not often used by TM systems because it allows these and similar, not always intuitive, executions.

Another appealing alternative to opacity is serializability with sandboxing. With this approach, the runtime intercepts the exceptions resulting from the execution of inconsistent transactions, such as the division by zero exception in Section 2.2, and does not propagate them to the user code. Instead, the transaction that throws the exception gets aborted and restarted. While intercepting the exception in the runtime solves problems with inconsistent transactions accessing deallocated memory or causing divisions by zero, it is not sufficient to eliminate all the problems their execution can cause.

Figure 2.9 shows pseudo-code of two transactions similar to the transactions in Figure 2.2. In this example, operation `OpB` loops based on the values of `x` and `y` instead of performing a division. If allowed to observe equal values of the two variables, which can only happen if the reads are not consistent, the operation loops indefinitely at line 16. To fully solve the problem of inconsistent reads, the runtime needs to periodically ensure that reads of executing transactions are consistent, in addition to intercepting the exceptions. The consistency of reads can be checked, for example, on every loop iteration or during garbage collection cycles.

Guaranteeing just serializability and using sandboxing to mask the problems caused by inconsistent transactions works well when STM is deeply integrated with the compiler and the runtime system. This makes it well suited to STMs for higher-level languages, such as Java and C#, but less so for STMs for lower-level languages, such as C/C++, on which I focus in this thesis.

Strong versus weak atomicity. The discussion so far implicitly assumed that all accesses to shared data are performed transactionally. For example, the discussion of opacity only considers transactions, assuming that all accesses to shared objects are transactional. The possibility of the same data being accessed by both transactional and non-transactional code is completely ignored. The simplest way to deal with non-transactional accesses to shared

data is to consider them as implicit, one-access transactions. Such a model is called *strong atomicity* and is typically ensured by HTM systems. With strong atomicity, the semantics of the TM are clear and well defined by opacity.

However, strong atomicity is rather cumbersome to fully implement in software, as it requires rewriting of non-transactional code to use one-access transactions for each memory access. Even with a compiler to aid with this task the overheads of executing all non-transactional memory accesses as short STM transactions can be prohibitive, despite possible compiler and runtime optimizations [102]. For this reason, STMs typically only ensure *weak atomicity* [79]. With weak atomicity, accesses to shared data are only permitted from transactions. If shared data are accessed from non-transactional code the results are not defined. With weak atomicity, opacity, as introduced, still defines correct STMs, as long as no objects are accessed by transactional and non-transactional code at the same time.

A special case of mixed accesses to an object occurs when the object transitions from being thread-local to being shared, which is known as the *publication* idiom, and when it transitions back, from being shared to being thread-local, which is known as the *privatization* idiom [106]. In these cases, the object is accessed non-transactionally only when it is private to a thread, and it is accessed from transactions otherwise. A typical example of object publication is when it is inserted into the shared data structure, and of privatization is when it is removed from the shared data structure. Privatization is sometimes used to optimize code: when an object is accessible only by a single thread, the thread can freely access it non-transactionally, thus reducing TM-related overheads and improving the performance. To fully provide the illusion that transactions execute as if they acquire a single lock, an STM needs to ensure both privatization and publication safety. Some STM implementation strategies, however, do not guarantee privatization and publication safety by default. The base algorithms of such STMs need to be altered to support these idioms, which sometimes results in significant overheads.

In the remainder of the thesis, I focus on STMs that provide weak atomicity and do not guarantee privatization and publication safety, unless stated otherwise. I focus on such STMs to evaluate the performance of the base STM algorithms, without having the evaluation impacted by, sometimes significant, overheads of supporting privatization and publication [106, 121]. Privatization idiom and the overheads of making an STM privatization-safe are discussed in more details in Sections 4.5.2 and 5.6.

3 Large Software Transactions

In this chapter, I present an experimental evaluation of several state-of-the-art STMs with STMBench7 [52], a large-scale STM benchmark, and discuss a number of rather surprising conclusions regarding the performance of STMs with large transactions based on these experiments. For example, it turns out that there is not as much difference in performance between obstruction-free and lock-based STMs with STMBench7 as with popular micro-benchmarks typically used for evaluation of STMs. Based on the results of the experiments, I draw conclusions about the best policies for supporting large transactions in STM. The design of SwissTM, presented in the next chapter, is largely based on these conclusions. Furthermore, I discuss several surprising findings regarding the robustness of the used STMs. In the experiments, all STMs crashed when executing with large transactions, revealing that they were not properly tested with large-scale workloads. This means that they are not likely to be well optimized for such workloads either. The evaluation also uncovered several programmability limitations of the library-based STMs, in particular related to the support for object-oriented language features and standard libraries.

3.1 Overview

Many of the recent STM proposals have been evaluated using well-known data structures, such as linked lists, red-black trees, hash-tables, and similar [28, 30, 43, 57, 62, 63, 69, 75–77, 92, 99, 105, 109, 115]. Whereas these micro-benchmarks can reveal performance differences between various STMs to a certain extent, they have an important limitation of being too small in scale. Thus, they do not provide sufficient insight into the behavior of STMs in programs with large transactions that access many, potentially complex, objects. One of the distinctive advantages of STM is the ability to execute transactions of arbitrary size, which makes such programs a particularly important target for STM.

STMBench7 [52], a more realistic benchmark for evaluating STM implementations, has been proposed recently. The main characteristic of STMBench7 is that it stresses the underlying TM in ways various micro-benchmarks do not: the large size of its data structure increases total

memory requirements, length of transactions, and the number of objects transactions access. Also, transactions exhibit non-uniform access patterns, further stressing the underlying TM. STMBench7 was first implemented in Java with ASTM [76] and two levels of locking. In this chapter I describe the experiments based on the C++ version of STMBench7 implemented with three STMs: RSTM [77], TL2 [28], and TinySTM [43]. I also discuss the issues encountered when implementing STMBench7 in Java with DSTM2 [62], which prevented me from experimenting with STMBench7 and DSTM2.

I base the experimental evaluation on these particular STMs for two reasons:

1. They are freely available and open-source, which allowed me to modify them when needed and inspect closely the reason for the observed behavior.

2. They cover a large part of STM design space, which allowed me to compare, for example: lock-based to obstruction-free design, eager acquire to lazy acquire, and invisible to visible readers, in the context of support for large transactions.

Next, I present an overview of the main conclusions resulting from stressing the used STM implementations with STMBench7.

Performance. STMBench7 provides a performance test that stresses STMs differently than common micro-benchmarks, which allows me to compare higher-level design choices instead of focusing on the lower-level ones. The micro-benchmarks typically execute small transactions on data structures of small sizes, which causes even very small overheads to have a significant relative impact. The small size of the transactions makes them a good choice for evaluating the lower-level aspects of the design, like the overheads of each STM read and write call or the amount of cache invalidation caused. In contrast, the impact of these lower-level overheads is not as high on larger transactions used in STMBench7, which enables a more meaningful comparison of the higher-level design. For example, STMs that are able to reduce the abort rate and the amount of work performed by the, subsequently, aborted transactions, at the cost of having more expensive read and write calls, might actually perform better than the simpler STMs that have faster reads and writes, but abort transactions too often, or too late. Another important consequence of using large transactions is their ability to fully expose all super-linear performance overheads in the STM, as the impact of such overheads is much higher with large transactions than with smaller ones.

The main conclusion of my experiments with STMBench7 is that the choice of conflict detection and contention management schemes has the highest impact on STM performance with larger-scale workloads. This contrasts previous conclusions, based on micro-benchmark experiments, that identified the low-level overheads, such as the cost of accessing a single object, as the main performance factor [29]. Based on the results, I identified that the conflict detection techniques that detect write-write conflicts early and read-write conflicts late have

the highest performance with large transactions over a range of contention levels. This is in contrast to work presented in [77], in which micro-benchmark experimental results did not favor any of the conflict detection variants, thus leading the authors to propose adaptive conflict detection. The examples of conflict detection schemes that are well suited to large transactions are the ones used by the RSTM variant that uses invisible reads with global commit counter heuristic and eager conflict detection, and TinySTM. I also identified an interesting relation between contention management and conflict detection, showing that with different conflict detection approaches different contention managers achieve the best performance. For example, Polka [100] contention manager performs better than Greedy [50] with lazy conflict detection, but Greedy is faster with eager conflict detection. These results are in contradiction with the previous study [100], based on typical micro-benchmarks, that favored Polka overall.

Robustness. With *all* used STMs, I encountered problems that lead to benchmark crashes, showing that the STMs could not cope with the STMBench7's large data structure and transactions. This means that, in practice, none of the used STMs were truly unbounded and dynamic. Some STMs incurred too high space overheads and could not fit in the memory, others overflowed internal data structures leading to either incorrect executions or crashes. These issues might not be surprising given that memory management is particularly difficult in STM environment: (1) STMs require more memory than regular programs for maintaining various kinds of internal logs, and (2) deallocating memory in environments without garbage collector is more difficult as the reachability of memory blocks must be computed before the deallocation. The fact that these STMs crashed when executing transactions of large sizes reveals that they were not extensively tested with large transactions, and, consequently, that their performance was not optimized for such transactions. It should be noted that, although the used STMs are not meant for production use, they were tested extensively and worked without problems in a number of micro-benchmarks. Consequently, I did not expect to find them crash, nor was that the goal of the experiments.

It is interesting to contrast STM with locking at this point. With locking, no extra memory is required for book-keeping purposes, which results in smaller memory requirements compared to STM. Also, all shared objects are protected with appropriate locks before being deleted, removing the need to compute reachability of objects before the deletion. Consequently, memory management with locking is much simpler and STMBench7 locking implementations work with standard memory allocator without problems.

Application programming. The unique features of STMBench7, when compared to other benchmarking approaches, are the use of: (1) external libraries that do not support STMs and (2) a bigger subset of object-oriented language features, such as polymorphism. This makes STMBench7 a good tool for testing programmability of STMs, as it highlights the problems that programmers might encounter when using STM with general, complex code.

The use of these constructs also highlights the costs of using STMs in terms of either limited programmability, in case when the STM does not support all the language constructs and libraries cleanly, or additional programming effort, it the programmers decide to re-implement parts of the required libraries themselves. For example, none of the used word-based STMs fully supports interfacing with external libraries, including the standard libraries, which considerably increases the cost of programming with these STMs. My experience from the experiments confirms the previous conclusions [23] that STM compilers are needed to truly deliver on promise of easy programming with STM, as the library-based STMs can be too cumbersome to use. Furthermore, transactional versions of standard libraries should be provided to allow programmers to use the same techniques and libraries as when writing sequential code.

To summarize, the experiments with STMBench7 revealed the following:

1. The choice of appropriate policies, such as the conflict detection and contention management techniques, has bigger impact on performance of large transactions than the cost of low-level mechanisms, such as the choice between lock-based and obstruction-free implementation.

2. The best conflict detection techniques detect write-write conflicts eagerly and read-write conflicts lazily.

3. With eager conflict detection, Greedy contention manager achieves the best performance with large transactions.

4. *All* used STMs crashed, in one way or another, mostly due to problems related to memory management, showing that they were neither tested nor optimized for large transactions.

5. Most of the tested STMs have serious usability issues, mainly with object-oriented features of the language and the external libraries. Support for these needs to be improved if STMs are to be more widely used.

It is important to point out that mentioning various problems with particular STMs does not have the purpose of bashing these STMs or their authors, but merely of highlighting challenges underlying support for large transactions in STMs. In fact, I promptly received help or fixes for most of the problems I reported to the authors, which enabled me to perform the presented experimental evaluation.

3.2 STM design space

The main task of an STM is to *detect* conflicts among concurrent transactions and *resolve* them. Deciding what to do when conflicts arise is performed by a conceptually separate component

called a *contention manager* [50, 63, 100]. There exist several different conflict detection and contention management policies. Besides the choice of these policies, STM design space consists of several other axes. Next, I briefly describe the main design choices when building an STM, focusing on the conflict detection and contention management.

3.2.1 Conflict detection

Most STMs employ the single-writer-multiple-readers strategy, where accesses to the same location by concurrent transactions conflict when at least one of the accesses is a write. In order to commit, each transaction must eventually *acquire* every object that it is updating. The acquiring can occur at the time of the first update access to the object, in which case it is called *eager*, or it can occur at commit time, in which case it is called *lazy*. These are also called eager and lazy writes, or updates.

Furthermore, when reading objects, transactions can employ either *visible* or *invisible* readers. With visible readers, transactions update the STM meta-data even when just reading the application data, to make their reads visible to other transactions. The updates of the meta-data are typically performed using expensive atomic CPU instructions and require additional memory barriers, which increases the cost of the reads. The updates of shared meta-data also increase cache contention, further degrading the performance of reads. In contrast, with invisible reads, the readers do not update the shared meta-data, but they have the sole responsibility of ensuring consistency of their reads, as required by opacity. To ensure the consistency of the reads, transactions typically check that none of the objects they have read so far was updated in the meantime by a different transaction. The set of the objects read by the transaction is called its *read-set* and checking that it is consistent is known as the read-set *validation*. Similarly, the set of the objects updated by the transaction is called the *write-set*. The time complexity of a basic validation algorithm is proportional to the size of the read-set, but can be improved with the global commit counter heuristic [108], or the time-based validation scheme [28, 92].

With the global commit counter heuristic, each update transaction increments the shared commit counter upon commit. The counter is used to avoid some of the read validations in the following way. When a transaction starts, it reads the commit counter. When it reads an object, it checks whether the commit counter has changed. If it has not changed, the transaction can safely skip the read-set validation, as nothing in the system has changed since the last validation of the read-set, thus guaranteeing that the read-set is still consistent. If the counter has changed, this means that one or more transactions committed recently, possibly updating some of the objects in the transaction's read-set. Therefore, the read-set has to be validated to ensure its consistency. The transaction can use the newly observed value of the commit counter for the future reads after the successful validation.

With the time-based scheme, each object is assigned a version number that is generated using a single shared counter: when a transaction commits, it increments the counter and sets the

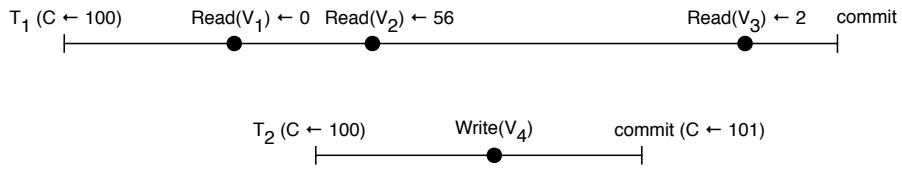


Figure 3.1: The difference between the global commit counter and the time-based schemes.

version of each updated object to the new value of the counter. When a transaction starts, it reads the shared counter, similarly as with the global commit counter heuristic. The value of the counter is called transaction's *validity timestamp*, and it represents the last known point at which transaction's read-set was valid. When the transaction reads an object, it compares the object's version to the transaction's validity timestamp. If the object's version is lower or equal to the validity timestamp, the validation can be omitted, as the object has not been updated since the last known point at which the read-set was valid. Otherwise, the transaction can either abort immediately or validate its read-set. If the validation succeeds, the transaction can adopt the new version of the shared counter as its validity timestamp and continue. The validation of the read-set and the update of the transaction's validity timestamp are known as the *read-set extension* [43]. I discuss a similar conflict detection scheme in more detail in the next chapter.

The time-based scheme is finer-grained than the commit counter heuristic as transactions avoid read-set validations even if the shared counter changes, as long as the objects they are actually reading have not been updated since the last validation. This difference is illustrated in Figure 3.1. The figure shows versions read from and written to the global counter on transactions' starts and commits and the version of each object transactions read. With the commit counter heuristic, transaction T_1 has to validate its read-set when reading V_3, as the value of the commit counter has changed since T_1's start. With the time-based STM, however, the validation is not necessary as versions of all objects accessed by T_1 are lower than its starting validity timestamp.

Other conflict detection approaches than the pure eager and lazy schemes exist. For instance, *mixed invalidation* [108] represents a mix between the pure lazy and pure eager scheme as it detects write-write conflicts eagerly and read-write conflicts lazily. A similar conflict detection scheme is provided by the more general, but also more expensive, *multi-versioning* techniques [16, 92].

Alternatively to the approaches discussed above, which typically assign versions to each object or memory location, several STMs rely solely on centralized meta-data when detecting conflicts among transactions [24, 86, 109]. The simplest such scheme uses a single lock to serialize all transactions during their execution [24]. Other approaches, for instance, serialize the commit phases of transactions using a single lock and rely on value-based validation of read-sets [24, 86]. Whereas these STMs reduce the costs of read and write STM accesses, they

are not well suited to large transactions as they serialize either whole transactions or their commit phases.

Regarding the STMs used in the experiments, RSTM supports four basic algorithm variants: it supports all combinations of lazy and eager acquisition, and visible and invisible readers. Furthermore, it can be configured to approximate mixed invalidation using the commit counter heuristic. TL2 and TinySTM use lazy and eager acquisition, respectively. Both TL2 and TinySTM employ invisible readers and rely on the time-based scheme to speed-up read-set validations.

3.2.2 Contention management

The contention manager decides what a given transaction, called the *attacker*, should do in case of a conflict with another transaction, called the *victim*. Possible outcomes are: aborting the attacker, aborting the victim, or forcing the attacker to wait and retry after some period. Here, I describe several contention managers used in the experiments, which are discussed in more detail in [50, 100]:

- *Timid* is the simplest contention management scheme which always aborts the attacker, possibly with a short back-off between the abort and the restart. This is the default contention management scheme in TL2 and TinySTM.
- *Polka* [100] assigns every transaction a priority equal to the number of objects the transaction has accessed so far. Whenever the attacker waits, its priority is temporarily increased by one. If the attacker has a lower priority than the victim, it will be forced to wait, using exponential back-off to calculate the wait interval, otherwise the victim gets aborted. Polka has been shown previously to provide best performance in smaller-scale benchmarks [100].
- *Greedy* assigns each transaction a unique, monotonically increasing timestamp on its start. The transaction with the lower timestamp always wins. An important property of Greedy is that, unlike other mentioned contention managers, it avoids starvation of transactions.[1]
- *Serializer* is very similar to Greedy except that it assigns a new timestamp to transactions on every restart. Therefore it does not prevent starvation of transactions, but it still prevents livelocks.[2]

Interestingly, several contention managers can also be combined at runtime [49]. As it is not clear what the best policy for dynamically switching between contention managers at runtime is, I did not use such a polymorphic contention manager in the experiments.

[1] Greedy avoids starvation only if visible readers are used.
[2] Serializer avoids livelocks only if visible readers are used.

3.2.3 Access granularity

The previous chapter makes a distinction between word-based and object-based STMs, based on the granularity of accesses exposed in the STM interface. The difference in the access granularity typically impacts the implementation of the STM as well. Word-based STMs perform conflict detection and logging at the granularity of memory words or groups of memory words, for example at the cache-line granularity, whereas object-based STMs perform them at the granularity of objects. Regarding STMs used in the experiments, RSTM is object-based while TL2 and TinySTM are word-based.

3.2.4 Update policy

Transactions can use direct or deferred updates, as described in the previous chapter. With the direct-update STMs, transactions directly update the shared objects at the time of writing and rollback the updates if they get aborted. With the deferred-update STMs, transactions buffer the updates in the transaction-local logs and apply them to the shared objects only during transaction commit. The choice of the update policy and the conflict detection approach is not completely orthogonal: with the direct updates, transactions have to eagerly acquire objects for writing, or, otherwise, concurrent transactions could not distinguish between reading tentative and committed values of objects. Regarding STMs used in the experiments, RSTM and TL2 use deferred updates, whereas TinySTM employs direct updates.

3.2.5 Progress guarantees

There are two general classes of STM implementations: lock-based and non-blocking. Lock-based STMs implement some variant of the two-phase locking protocol [42]. Non-blocking STMs [63] do not use any blocking mechanisms, such as locks, and can thus ensure progress even when some of the transactions are delayed or even crashed. Most of the non-blocking STMs are obstruction-free [63, 75–77, 115], although lock-free STMs have also been implemented [46, 54, 101]. Regarding STMs used in the experiments, RSTM is obstruction-free, whereas TL2 and TinySTM internally use locks. The lock-free STMs are not represented in the experiments, as they are considered to have lower performance than the obstruction-free ones.

For a more complete survey of STM techniques see [55].

3.3 STMBench7

Next, I present STMBench7 and its implementation. I start with an overview of several alternatives to STMBench7 which are commonly used for STM benchmarking, and then dive into the details of STMBench7 itself.

Benchmark	Data size	Name	Tx Size	Tx/s
micro-benchmark	128	Linked list	64	10^6
		Hash table	2	10^6
		RB tree	7	10^6
STMBench7	700,000	LT on	100,000	10^1
		LT off	large 10,000 avg ~ 100s	10^3

Table 3.1: Comparison of STMBench7 and average micro-benchmark sizes.

3.3.1 Alternatives to STMBench7

The prevailing way of measuring the performance of STMs is by using micro-benchmarks, with the main goal of testing low-level details of STM implementations. Simple operations on simple data structures of modest sizes serve this purpose well, as there is no application work to mask TM implementation overheads. However, the simple structure of the micro-benchmarks is also their major limitation, as some of the design choices that look justified with micro-benchmarks might actually result in worse performance with more complex data structures and operations.

Besides STMBench7, several more realistic benchmarks have been considered, including SPLASH-2 [119], STAMP [17], Lee-TM [8], QuakeTM [47], Atomic Quake [123], WormBench [122], RMS-TM [68], and SynQuake [73]. I return to STAMP and Lee-TM in more detail in the next chapter, where I use them for a comprehensive evaluation of SwissTM on a wide range of workloads, and describe the remaining benchmarks in Chapter 6. I did not use the other benchmarks in the experiments either because they are developed in a different language, such as WormBench, which is developed in C#, or they require an STM compiler, such as QuakeTM, Atomic Quake, and RMS-TM. An important characteristic of STMBench7 is that its workloads use transactions that are larger than in any other STM benchmark I know of, thus providing insight into STM performance in unique scenarios.

3.3.2 Data and operations

STMBench7 [52] is specifically targeted at benchmarking STMs. Its data structure and operations, in large part inherited from the OO7 [18] benchmark for object-oriented databases, represent a workload typical for CAD/CAM/CASE software. This means that, although CAD/CAM/CASE applications are probably not a typical target for the future STM applications, STMBench7 workloads correspond to realistic, complex, object-oriented programs and, as such, represent very important target for STMs.

STMBench7 exhibits a large variety of operations, ranging from very short, read-only operations to very long operations that modify large parts of the data structure. STMBench7 also defines different workloads, ranging from static, read-dominated workloads with low contention between threads to dynamic write-dominated, high contention workloads. The

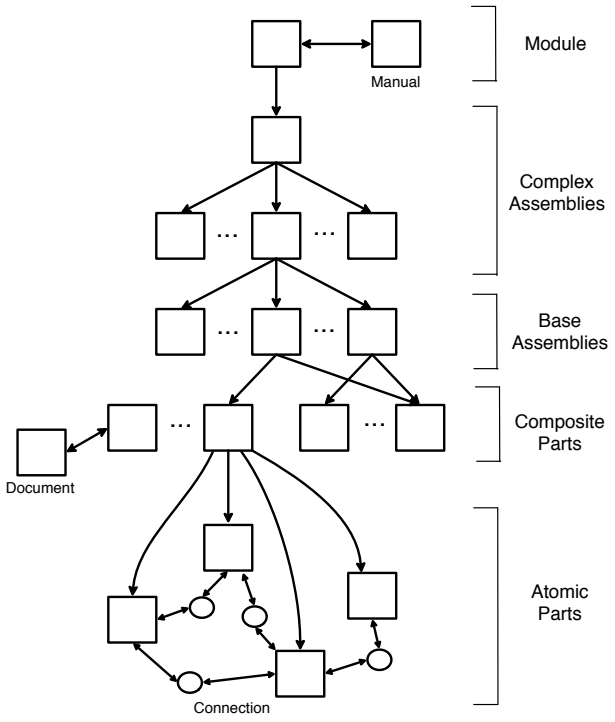


Figure 3.2: STMBench7 data structure.

data structure used by STMBench7 is many orders of magnitude larger than in typical micro-benchmarks. Consequently, its transactions are also longer and access more objects. The large size and variety of the STMBench7 transactions stress STMs in different ways than micro-benchmarks do. Thus STMBench7 can be, as experience shows very effectively, used as a performance, crash, and programmability test for STMs. The difference between sizes of data structures and transactions in STMBench7 and typical micro-benchmarks is highlighted in Table 3.1. The table reports data and transaction sizes in numbers of objects, giving sizes of typical micro-benchmarks with the value range of 256 and sizes of STMBench7 transactions with and without long traversals.

A run of STMBench7 consists of creating a randomized data structure and executing a random mix of operations on it, with each operation executed as a separate transaction. STMBench7 uses a tree-like data structure depicted in Figure 3.2. The data structure consists of a single module object which has an associated manual. The module is connected to a tree of complex assembly objects. There are six levels of complex assemblies, which are, at the last level, linked to the base assemblies. Each base assembly is connected to a number of composite part objects, that form a shared design library. Composite parts have private graphs of atomic parts that are, in turn, connected to each other via connection objects. Each atomic part also has an associated documentation object. STMBench7 defines six indexes to index the various object types, enabling the operations to start traversing the data structure at different starting points. The operations traverse the data structure in different directions too, moving up or down the

	Workload type		
Category	Read	Read-write	Write
Read-only ops	90	60	10
Update ops	10	40	90
Long Traversals	5		
Short traversals	40		
Short operations	45		
Structure mods	10		

Table 3.2: Default ratios of executed operations in percents.

tree, or accessing randomly selected objects. These varied access patterns preclude usage of simple locking techniques and also stress the ability of the STM to deal with deadlocks and livelocks.

There are four main categories of STMBench7 operations:

1. *Long traversals* are operations that access large parts of the data structure, typically all assemblies and atomic parts.

2. *Short traversals* access a much smaller number of objects, traversing the data structure along a random path, starting from a module, atomic part, or a document. Some of the short traversals use the indexes.

3. *Short operations* perform simple operations on a randomly selected object or its neighbors. Some of the operations use indexes to search for objects using various search criteria.

4. *Structural modifications* change the data structure by creating or deleting objects or links among them. Structural modifications are performed in a way that prevents significant degeneration of the data structure, such as, for example, making parts of the structure disconnected from the root. The maximum size of the data structure is also constrained.

Operations in the first three categories can be either read-only or update operations. A distribution of executed operations is determined by the selected workload type, which can be read-dominated, read-write, or write-dominated. In addition to selecting a desired workload type, long traversals can be included or excluded from the workload. This, in essence, results in six different workloads. The distributions of operations for different workloads are given in Table 3.2.

STMBench7 defines two different locking schemes: the coarse-grained scheme, which uses a single read-write lock to protect the whole data structure, and the medium-grained scheme, which uses a handful of locks to increase the parallelism between concurrent operations. The medium-grained scheme protects each of the levels in the data structure with a different

read-write lock, and uses a separate lock for structural modifications to isolate them from all other operations.

One peculiar characteristic of STMBench7 is the data structure initialization, which is performed in a single transaction. The initialization transaction is not included in the measurements, but, it turns out that several STMs could not execute it correctly due to its large size. Therefore, the initializing transaction makes an important component of STMBench7 as a correctness test.

3.3.3 STMBench7 with word-based STMs

STMBench7 is inherently object-based, as it uses class inheritance and polymorphism extensively. Furthermore, it relies on standard language libraries, such as C++ STL and standard Java library. In order to implement STMBench7 with word-based STMs, like TL2 and TinySTM, I implemented a thin object-based wrapper on top of their word-based interfaces, as depicted in Figure 3.3. The wrapper is, in fact, an implementation of an object-based STM on top of a word-based one.

The interface of this “composite" STM is very similar to the interface of RSTM, enabling the reuse of the bulk of the code of benchmark's initial C++ implementation, which was done with RSTM. The interface relies on C++ smart pointers to eliminate some of the incorrect uses of the object-based STM interface [23]. In short, it provides three smart pointer types: `sh_ptr`, `rd_ptr`, and `wr_ptr`. The pointers to objects that have not been opened for reading or writing are represented by `sh_ptr` objects. No function can be invoked on these pointer types. To perform a read-only operation on an object, its corresponding `sh_ptr` needs to be typecast into a `rd_ptr`. During the conversion the object is opened for reading. A `rd_ptr` pointer only supports read-only accesses, effectively being equivalent to a `const` pointer in C++. To perform update operations on an object, its `sh_ptr` needs to be typecast into a `wr_ptr`, which supports both read and update accesses. A fourth pointer type, `un_ptr` is also provided, to enable performing operations on the objects outside of transactions. Whereas the smart pointer interface makes programming less error-prone by detecting some erroneous uses of the STM's interface at compile time, it still does not eliminate the need for the compiler support [23]. My experience, presented in Section 3.6 confirms this.

The wrapper internally represents each object by its handle, which is a pointer to the current version of the object. The handle of the object is accessed using the underlying word-based STM, which detects and resolves conflicts among concurrent transactions, while the wrapper manipulates object versions. Whenever a transaction opens an object for writing, a redo copy of the object is created and its address is speculatively written to the object's handle using the word-based STM. If the transaction commits, the redo copy becomes the current object version, as the address of the redo copy gets committed to the handle by the word-based STM. On the other hand, if the transaction aborts, the word-based STM rollbacks the change of the handle, which then keeps pointing to the object's old version.

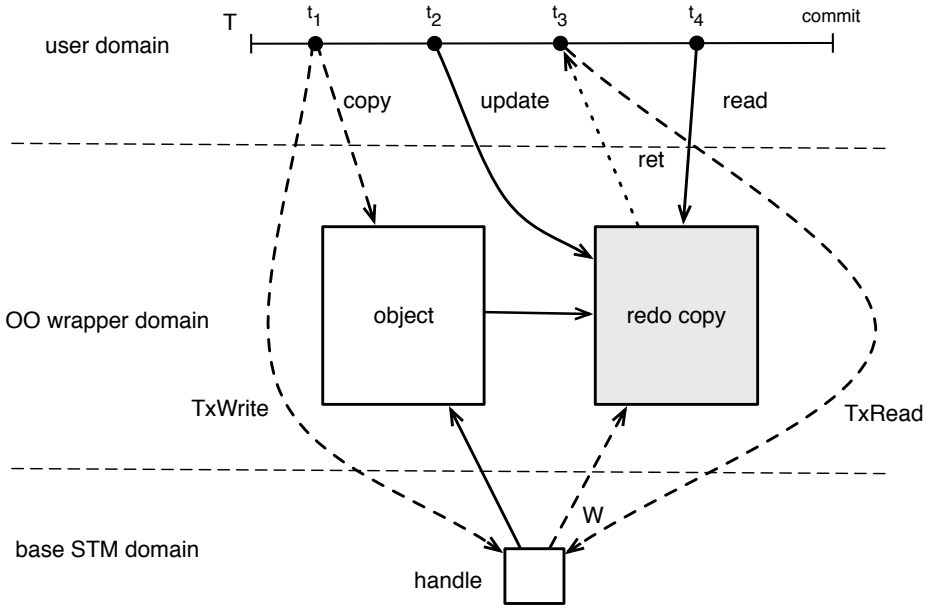


Figure 3.3: Implementing an object-based interface using a word-based STM.

The wrapper maintains a log of all objects opened for writing by the transaction. When the transaction commits, it uses the log to delete old versions of all updated objects. The old values of objects are deleted using transactional deallocation, which is described below. If, on the other hand, the transaction aborts, it uses the log to delete their redo copies. When opening an object for reading, the transaction reads the object's handle using the underlying word-based STM, which returns the pointer to the correct object version, avoiding read-after-write hazards. With the described approach, a handle always maps to the same object, hence its address uniquely identifies the object during its lifetime. The actual contents of the object reside at different memory locations as the new versions of the object replace the old ones.

The example depicted in Figure 3.3 illustrates what happens when transaction T accesses some shared object. T opens the object for writing at time t_1. To do so, it creates a redo copy of the object and writes its address to the object's handle using the word-based STM. After this point, T can access the redo copy of the object by reading its handle, whereas other transaction still get the address of the current version when they read the handle.[3] T next updates the contents of the object's redo copy at time t_2. When T subsequently opens the object for reading at time t_3, it reads the handle using the word-based STM. The read returns the address of the redo copy, thus correctly handling read-after-write by T. T actually reads from the redo copy at time t_4. Once the transaction commits, the old value of the object is discarded and the redo copy becomes the current object version.

[3] Assuming that the underlying word-based STM detects read-write conflict lazily and allows transactions to read memory locations that are being concurrently updated by other transactions.

It is worth noting that the wrapper supports various designs of the underlying word-based STM, as it only requires the word-based STM to guarantee opacity. Hence, it works with both TL2 and TinySTM, as well as with SwissTM presented in the next chapter.

Besides object versioning, the wrapper also deals with memory management. It rolls back allocations on transaction aborts and postpones object deletions until they can be performed safely. I adopted a memory management approach similar to that of [46], McRT malloc [66], and RSTM [77]. When a transaction allocates an object, the allocation is logged. If the transaction aborts, all the objects it allocated are deleted, which effectively rolls back the allocations. Similarly, when a transaction deletes an object, the deletion is logged and postponed until the commit. However, the object cannot be deleted immediately on commit, as the concurrent transactions might still hold references to the object and may try to access it in the future. To prevent accesses to deallocated memory, the object deletion is postponed as long as the object is reachable from any live transaction.

To determine when it is safe to delete objects, the wrapper maintains per-thread counters which count the number of transactions executed by each thread. The counters are incremented both at the start and at the end of each transaction. When a transaction commits, it stores all the objects it deleted and the current values of the counters into a deallocation log. The objects get actually deleted only when all the transactions that are concurrent to the deallocating transaction complete. The per-thread counters are used to determine when this happens: once the counters of all threads change, the in-progress transactions are completed and it is safe to deallocate the object. Note that there is no need for a transaction to snapshot the counters. The counters are monotonically increasing so using inconsistent counter values might postpone object deallocations, but does not compromise the correctness. I optimize memory deallocation by performing these reachability tests infrequently and for batches of objects at a time. More details about memory management with STM are given in Section 4.2.4.

Although the wrapper introduces additional overheads to word-based STMs, it enables the straightforward use of TL2 and TinySTM with C++ and non-transactional libraries. The wrapper's overheads are comparable to version management overheads of RSTM, thus allowing for a more accurate performance comparison of lock-based and obstruction-free STMs.

3.4 Performance results

In this section, I present the most interesting results of the experiments with STMBench7 and the conclusions about the best STM design for large transactions.

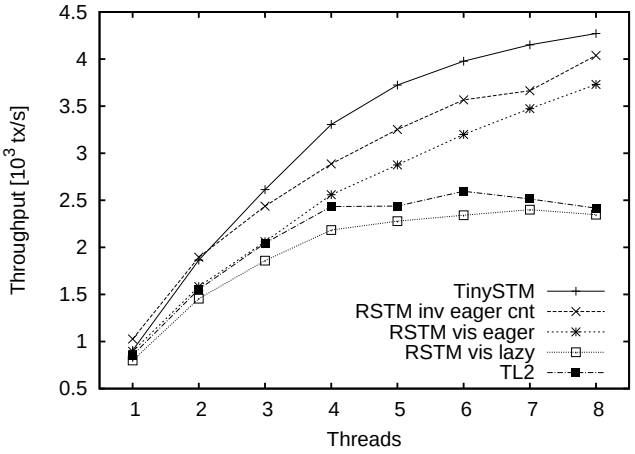


Figure 3.4: Comparison of different conflict detection approaches.

3.4.1 Experimental settings

All experiments were performed on a system with four AMD Opteron 8216 processors. These processors are dual-cores, providing eight cores for the experiments in total. Each CPU is clocked at 2.4 GHz. The system has 8 GB of memory. I used the highest compiler optimization settings of GNU g++. All presented results are averaged over multiple runs, where the length and the number of runs were chosen to reduce variations in the collected data. Unless stated otherwise, the results present read-dominated STMBench7 workloads without long traversals. The results for the read-write and write-dominated STMBench7 workloads are omitted, as the performance trends on these workloads and the read-dominated workload are similar. The following versions of the STM libraries were used: RSTM v3, x86 port of TL2 version 0.9.2, and TinySTM version 0.7.1. The STMs were configured to maximize their performance.

3.4.2 Locking versus obstruction freedom.

Previously published results suggest that lock-based STMs outperform the obstruction-free ones [29, 41]. My experiments that compare the performance of lock-based TL2 and TinySTM with the obstruction-free RSTM confirm this to a large extent. However, the choice between the obstruction-freedom and locking turns out not to have the greatest impact on the performance of STM. Figure 3.4 shows that, with comparable conflict detection approaches, locking outperforms obstruction-freedom: TinySTM performs better than the RSTM configured to use invisible reads with eager acquire and global commit counter heuristic, and TL2 outperforms RSTM with lazy acquire. The figure, however, reveals that the obstruction-free STMs can perform better than sub-optimal locking STMs, as the best RSTM configuration outperforms TL2 by a significant margin. The reason for RSTM outperforming TL2 lies mostly in its early detection of write-write conflicts, as discussed below.

3.4.3 Towards the ideal conflict detection approach

The best performance in the STMBench7 experiments was achieved when using conflict detection policies that have similar characteristics: they avoid wasted work performed by transactions that are likely to abort, but they also avoid aborting a transaction as long as there is some chance that it might commit. In a sense, these conflict detection policies are rather optimistic, but not too optimistic. The best performing RSTM variant uses invisible reads with eager acquire and the global commit counter heuristic, as shown in Figure 3.4. It has the best performance precisely because it detects write-write conflicts early, so transactions doomed to abort due to a conflict do not perform any further work in vain as is often the case with lazy conflict detection, but postpones the detection of read-write conflicts, thus allowing more parallelism than, for example, RSTM with visible readers. A similar conclusion can be drawn when comparing the two lock-based STMs in the figure: TinySTM detects write-write conflicts earlier than TL2 and this is, in a big part, why it outperforms it by a large margin.

I conjecture that the next step towards the ideal conflict detection policy is to keep detecting write-write conflicts eagerly, but to try to allow even more parallelism between concurrent transactions. In terms of the used conflict detection policies, both TinySTM and the best RSTM variant could be improved by allowing two conflicting transactions, where one reads and the other writes the same object, to proceed even after the reader detects the conflict, until it is certain that one of the transactions has to abort. With such a conflict detection scheme, both transactions would be allowed to commit in cases when the reading transaction commits first, thus avoiding the waste of work performed by the reader. Having multiple versions of the same object, as suggested in [16] and [92], could also help as it allows more transactions to commit, particularly the read-only ones.

3.4.4 Visible reads

The conflict detection schemes that use visible reads, which are commonly considered to have low performance, perform quite well in STMBench7 experiments: RSTM configured to use visible reads with eager acquire is the second best performing RSTM configuration and third overall, as shown in Figure 3.4.

Previously published results, based on the experiments with micro-benchmarks, have shown that STMs employing invisible reads outperform the ones that use visible reads in almost all cases, even when read-set validations are not optimized using the commit counter heuristic or the time-based scheme [77]. The low performance of visible reader schemes is attributed to higher rates of cache invalidations, resulting from frequent updates of the meta-data, and generally higher cost of reads. However, in my experiments with STMBench7, the quadratic cost of incremental read-set validations, incurred by invisible readers, becomes overwhelming with long transactions. Consequently, RSTM variants that use invisible reads without the commit counter heuristic have considerably lower performance than RSTM variants with visible reads. In the extreme case of STMBench7 workloads with long traversals, RSTM with

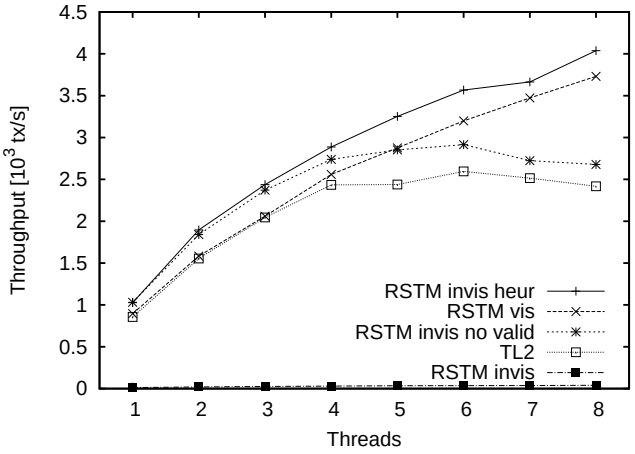


Figure 3.5: Incremental validation cost with RSTM.

invisible readers does not complete a single long traversal in minutes when the commit counter is not used. The results without the long traversals are presented in Figure 3.5. They show that, even without the long traversals, the RSTM variant with visible readers outperforms the variants with invisible readers by up to two orders of magnitude.

To confirm that the reason for the low performance of invisible reads is, in fact, the cost of incremental validations, I modified RSTM's code and turned the incremental validations off. The results presented in Figure 3.5 confirm that the RSTM performance with invisible reads without incremental validation approaches the performance with visible reads. This experiment also confirms that STMs that do not ensure opacity, as is the case with the modified invisible-read RSTM that does not perform incremental read-set validations, indeed encounter the problems discussed in Section 2.2, as the benchmark crashed several times due to memory corruption during the experiments.

Figure 3.5 shows that RSTM with invisible reads does not outperform RSTM with visible reads even when incremental validations are turned off, which might come as a surprise. The main reason for RSTM with visible reads still outperforming the RSTM with invisible reads is that it actually implements a better conflict detection policy: it detects transactions doomed to abort early, thus reducing the amount of wasted work, unlike RSTM with invisible reads that postpones read-write conflict detection until the commit time of the reader. However, RSTM with visible reads detects read-write conflicts too eagerly, which limits the parallelism and results in a slightly lower performance than when using invisible reads and the commit counter heuristic.

3.4.5 Towards the ideal contention manager

Similarly to the best conflict detection policies, the best performing contention managers avoid aborting transactions as much as possible. Furthermore, if one of the transactions

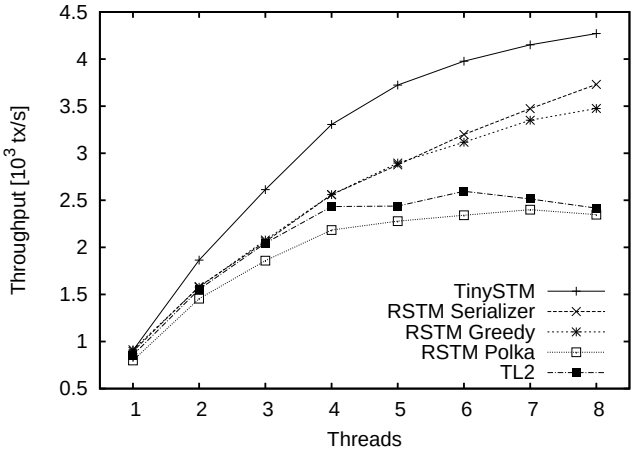


Figure 3.6: Performance of different contention managers.

needs to be aborted, they choose the transaction that performed the least amount of work. Figure 3.6 conveys the best performance achieved with several contention managers. Two best performing RSTM contention managers are Serializer, as implemented in RSTM [77], and Greedy [50]. These contention managers order the transactions by having them acquire unique timestamps at the start. They prioritize transactions that have lower timestamps, and have, thus, started earlier, which is a good approximation of the amount of performed work. With their approach, the “younger" transactions never abort the “older" ones, and instead wait for them to finish. On the other hand, the “older" transactions always abort the “younger" ones and are, thus, able to progress.

Polka was previously shown to perform well across a range of micro-benchmarks [100], but it does not achieve the best performance with STMBench7. Although Polka indeed aborts transactions that performed less work, it sometimes aborts transactions before that is absolutely necessary. This happens because the attacker waits for a limited time before aborting the victim even if the victim performed more work. With sub-optimal setting for this time limit, Polka sometimes aborts the transaction that performed more work, thus wasting more work than necessary.

An interesting contention management policy is the one employed by TinySTM, called Timid. In locking STMs, especially in the ones with eager acquire and direct updates like TinySTM, it is not trivial to abort the victim, as the abort involves releasing all the locks the victim has acquired and rolling back all its tentative changes to the memory. For this reason, lock-based STMs often simply abort the attacker and make it back-off for a while before retrying the transaction. Interestingly, the version of TinySTM I used does not force transactions to wait at all: it simply aborts the attacker and retries it immediately whenever a conflict is detected. Perhaps surprisingly, it turns out that this approach works quite well, as conveyed by Figure 3.6, despite causing a huge number of aborts. The Timid contention manager approximates the ideal contention management strategy in cases when transactions access the shared objects

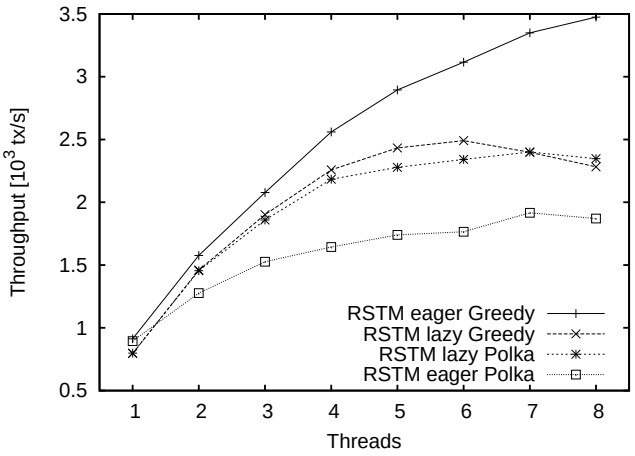


Figure 3.7: Performance of RSTM with different combinations of conflict detection and contention management policies.

in roughly the same order, which occurs quite often. When objects are acquired in the same order by many transactions, the transaction that acquires the first of these objects, is typically also the transaction that started executing the earliest and has, thus, performed the most work.

3.4.6 Conflict detection and contention management

The experimental results reveal a strong relation between the conflict detection and contention management policies. Figure 3.7 shows that RSTM with Polka performs better when using the lazy than when using the eager conflict detection. On the other hand, RSTM with Greedy performs better with the eager than with the lazy conflict detection. Similar observations can be made for other contention managers. These results demonstrate that comparing several contention managers while fixing a particular conflict detection approach, or vice-versa, does not necessarily yield a meaningful comparison. Stated differently, the construction of a new contention manager requires prior knowledge of the conflict detection policy it will be used with.

3.4.7 High concurrency levels

Running the benchmark with more threads than there are CPU cores in the system results in unexpectedly sharp performance degradation in several cases, as illustrated in Figure 3.8. With more threads than cores, performance cannot keep improving as there are simply no more processing resources available, but it should degrade gracefully. The graceful performance degradation is especially important once STMs reach ordinary users, as they cannot be expected to fine-tune configurations of their applications according to the number of available cores in the system, even if such approach would be feasible in some scenarios.

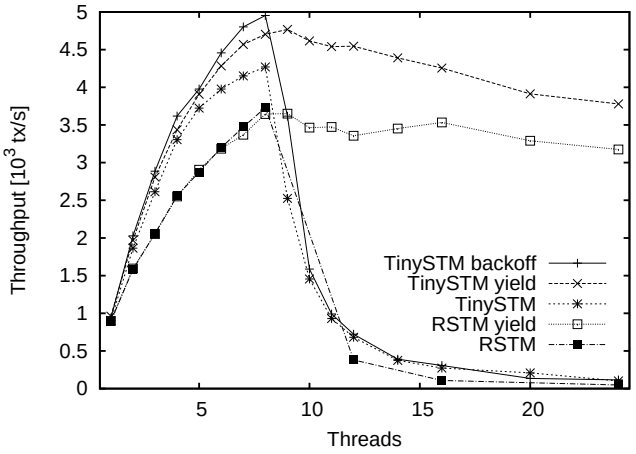


Figure 3.8: Performance of preemptive and non-preemptive STMs at high concurrency levels.

The main reason for this significant performance impact is the busy waiting performed by transactions. Busy waiting wastes processors cycles, but even worse, it makes logically blocked threads seem busy to the operating system scheduler. Because of this, the scheduler keeps assigning cores to these threads, possibly preempting the ones that can actually make progress. Such scheduling decisions slow down the system even further. This sharp performance degradation when the system is overloaded is not obvious with short transactions of micro-benchmarks, for the following reason. The busy waiting ends as soon as the transaction can make progress, which happens when the conflicting transaction commits or aborts. When transactions are short, the waiting period is short as well and busy waiting does not waste as much processor time as with longer transactions. The busy waiting can be even beneficial with short transactions, as it reduces the “reaction" time of the waiting threads by eliminating the overheads of thread switching.

I first noticed this problem when using RSTM with the Serializer contention manager. To test whether the busy waiting was indeed the cause of the performance degradation, I replaced the busy waiting with a call to `yield` function. The call to `yield` makes the scheduler assign the thread's CPU to a different thread, if there is one ready to execute, making it the simplest form of real waiting. As Figure 3.8 demonstrates, this simple approach is enough to solve the problem.

I also observed the similar sharp performance degradation on an overloaded system with TinySTM. The reason was pretty much the same: by aborting and restarting transactions immediately, the thread is effectively busy waiting and preventing the other threads from using the CPU core it is executing on. I applied the same simple solution as with the RSTM, by having each aborted transaction invoke `yield` before restarting. As the figure conveys, this was again enough to solve the problem.

Interestingly enough, invoking `yield` in transaction restarts in TinySTM resulted in a slight performance improvement even with thread counts lower than the number of the CPUs in the

Crash	STM	Cause
Memory management	RSTM v2	16 MB/thread limit
	DSTM2	high overheads
	TinySTM	free on commit
Transaction size	RSTM v3	1 MB/object limit
	TL2 x86 0.9.0	internal overflow

Table 3.3: Summary of observed bugs.

system. This performance improvement is the result of short back-offs induced by a call to `yield` even when no thread switch happens. These back-offs reduce contention, also reducing the abort rate. I verified that this interpretation of the results is correct, by implementing a simple busy waiting back-off mechanism in TinySTM. The results in Figure 3.8 confirm that the short explicit back-offs on aborts indeed improve the performance of TinySTM. A similar, beneficial effect of expensive actions performed on aborts was also reported in [96].

3.5 STM robustness

The experiments I performed demonstrate the effectiveness of STMBench7 as a correctness testing tool for STM implementations: I discovered several bugs in the STMs I used, that lead to incorrect program executions or crashes. The two main reasons for the crashes were: (1) memory management related problems and (2) the inability of STMs to cope with large transactions. Both of these are rooted in the large size of STMBench7 data structure, which stresses memory manager, by requiring large amount of memory, and implies large transactions. The encountered problems are summarized in Table 3.3.

3.5.1 Memory restrictions

As already pointed out, and somewhat surprisingly, I discovered that some of the STMs used in the experiments could not cope with the large memory requirements of STMBench7, or could not adequately handle memory deallocations. Inadequate handling of memory deallocations resulted in subtle problems with support for non-transactional accesses to data. In these cases, I had to fix the bugs in STM implementations before running the experiments to collect the performance measurements presented in the previous section.

RSTM. As soon as I started porting STMBench7 to C++, I immediately encountered a problem of exhausting all available memory with RSTM v2 [77]. RSTM v2 comes with a custom memory allocator, that has an arbitrary limit of 16 MB of memory that can be allocated per thread. Given much higher memory requirements of STMBench7, this memory limit prevented the data structure initialization from completing. RSTM v3 comes with a flexible memory management module that enables programmers to use a wider range of memory

allocators, including the allocator from the standard library. Using the standard allocator resolved the problem.

This bug, although quite easy to detect and fix, caused no problems with the micro-benchmark tests, simply because none of them required allocating 16 MB of memory in a single thread. Therefore, it remained hidden even after testing RSTM with several micro-benchmarks.

DSTM2. DSTM2 [62] is written in Java, and, therefore, it uses Java's built-in garbage collector which relieves the developers from most of the memory management concerns. DSTM2 is an experimental library that makes it easy to implement various STM algorithms inside the framework it provides. The programers can transparently choose between different STM algorithms at run time. To support the transparent choice of the STM algorithm, DSTM2 generates various classes at run time: it generates a "transactional" version of every class of shared objects and also wraps every getter and setter method of every shared object in a separate object.

The first problem I encountered with DSTM2 was the exhaustion of memory that JVM uses to store class definitions. This was due to DSTM2 creating one transactional class for each instance of the original class, although a single transactional class would suffice. Due to the large number of objects in STMBench7, DSTM2 generated more classes than supported by the JVM, crashing it. After reporting this bug to the DSTM2 authors, I received a new version of the library that reused the class definitions between instances of the same class, which solved the problem.

However, I was still not able to allocate STMBench7's data structure, this time due to the lack of heap memory. The problem persisted even after increasing the JVM heap size to 8 GB. After further inspection, I found out that DSTM2 creates two wrapper objects for every data member of the object: one for the getter and another one for the setter function. Both of these wrappers hold a reference to a different instance of the `java.lang.reflect.Method` object, the size of which is an order of magnitude greater than the size of the rest of the object. With the large number of objects used in STMBench7, these `java.lang.reflect.Method` objects exhausted the heap. I further modified DSTM2 to have all wrapper objects reuse the same `java.lang.reflect.Method` instance, thus reducing memory requirements.

Whereas the memory requirements were reduced, the problem was still not fully solved, and I was still not able to run STMBench7 with DSTM2. The reason for the heap exhaustion even after these modifications was that DSTM2 still introduced big per-object overheads in its internal structures, which caused the STMBench7 initialization transaction to exhaust the heap. Fixing this problem turned out to require a significant rework of the library, which is out of the scope of this thesis. Therefore, I was not able to experiment with STMBench7 and DSTM2.

In contrast, micro-benchmarks do not create nearly as many objects as STMBench7, so the high bloat introduced by DSTM2 was overlooked. It is worth mentioning that the lock-based version of STMBench7 implemented in Java can be executed without any problems, as it has modest memory requirements of approximately 200 MB, which is far from the heap's size.

TinySTM. TinySTM uses a simple and efficient approach to memory management that works correctly when shared data is accessed only with STM read and write calls. The approach relies on a non-faulting load instruction, which is simulated using the appropriate signal handlers on architectures that do not support the instruction natively. Memory deallocations are performed immediately when the deallocating transaction commits. Before committing, the transaction acquires all locations in the deallocated block for writing, thus preventing concurrent transactions from writing to the deallocated block. The concurrent transactions could still read locations in the block, but the simulated non-faulting load instruction prevents crashes in these cases, and TinySTM ensures that the reading transaction is aborted subsequently. This approach is different from the memory manager in the STMBench7 object-based wrapper and similar approaches [46, 66, 77] which postpone the deallocation until all concurrent transactions have completed.

Unfortunately, when I tried to use TinySTM's memory manager instead of the wrapper's, STMBench7 crashed. The reason for the crashing was wrapper's caching of the pointers to the opened objects for the duration of the transaction. This caching is done to eliminate repeated calls to the underlying word-based STM and, thus, improve performance. The cached pointers allow concurrent transactions to access the objects that have already beed deallocated. Illegal memory accesses are handled by the simulated non-faulting loads in most cases, but they cause memory corruption in rare situations when the deallocated memory gets reused by a different thread. Long transactions of STMBench7 increase the probability of triggering the bug, as with long transactions the pointer to the old version of the object gets used long after the object deallocation, thus increasing the probability of a different thread reusing the memory.

This problem most often manifested itself while copying large C-style strings. In cases when zero-terminator of the string gets overwritten after the memory is reused, the string copying based on the cached pointer proceeds past the end of the string, overwriting memory allocator meta-data and corrupting the heap. Heap corruptions result in subsequent `malloc` and `free` calls raising an exception and aborting the program. I detected the problem only in these, relatively rare, cases when the heap became corrupted.

The problem is not obvious as it seems that the non-faulting load instruction should prevent heap corruptions. Long transactions of STMBench7 were instrumental in detecting the problem, as they increase the probability of incorrect executions and enabled me to discover that TinySTM's memory manager was being used incorrectly. I solved the problem by using the object-based wrapper's memory allocator, described above.

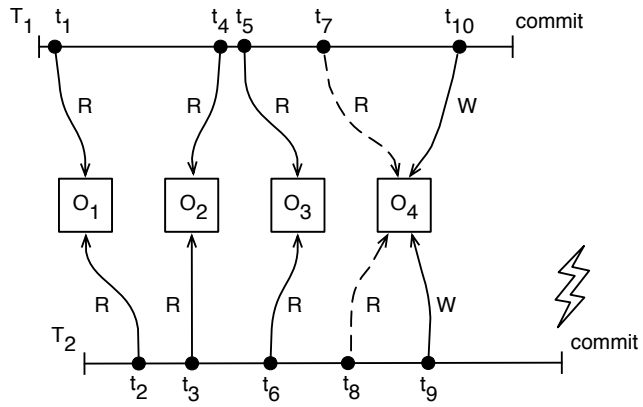


Figure 3.9: TL2 x86 version 0.9.0 read-set overflow example.

3.5.2 Transaction size

While running the benchmark, I encountered several problems that were caused by the size of executed transactions. In these cases, the problems were triggered by the high number of transactional accesses, not by the size of the accessed objects, typically as performing many accesses resulted in overflows of transaction's internal data structures.

RSTM. As mentioned above, RSTM v3 comes with an updated memory manager that supports several different underlying memory allocators, including the standard `malloc` implementations and the new version of RSTM's custom allocator. Despite the RSTM's custom allocator removing the 16 MB allocation limit per thread, the initialization transaction still crashed when using it. By investigating the problem, I discovered and fixed a pretty interesting bug that caused the crashes. RSTM's allocator allocates memory in chunks that are at most 1 MB in size, which limits the maximum size of objects. As there are no objects larger than 1 MB in STMBench7, this limit did not cause a problem when allocating STMBench7 objects. However, RSTM internally uses arrays for storing transaction's read- and write-sets, which dynamically grow to accommodate for the increasing size of the read- and write-sets as the transaction keeps accessing new objects. With large enough transactions, these arrays can become larger than 1 MB, which is the maximum object size supported by the memory allocator. Crossing this threshold causes the program to crash, practically limiting the number of objects that can be accessed inside a transaction.

Of course, the micro-benchmarks did not reveal this problem, as their transactions never access enough objects to overflow a 1 MB array. The problem was also not triggered by the allocation of STMBench7 objects, as none of them is larger than the 1 MB limit either. However, the long initializing transaction of STMBench7 crashed, not allowing me to use the custom memory allocator for running STMBench7, before fixing the bug. To discover the problem was, again, harder than to fix it and a small patch to RSTM's custom allocator was enough.

TL2. The problem with the number of accessed objects in a single transaction also occurred with TL2 [28] x86 version 0.9.0 [81]. TL2 maintains its read-set as a linked-list. It pre-allocates a number of the list nodes during STM initialization, for performance reasons. When a transaction uses up all of the available pre-allocated list nodes, it is aborted and restarted. Before restarting, additional list nodes are allocated to accommodate for the increase in the number of accessed locations. However, due to a programming error, transactions did not get aborted when overflowing the list. Instead, they were allowed to continue until the commit time, and completely ignored the reads causing the overflow when validating their read-sets. As a result, two conflicting transactions could be allowed to commit. The problem is illustrated in Figure 3.9. In the figure, I assume that initially only three list nodes are pre-allocated. As the list overflows, transactions T_1 and T_2 are both allowed to commit, despite them both reading and updating object O_4. This violates transactional semantics and can even crash the program in specific cases, which is how I initially noticed the problem.

When two transactions delete the same object they conflict as they both unlink the object from the shared data structure. However, if the conflict passes undetected, both transactions will actually delete the same object, causing a double deallocation of the occupied memory block. Most of the time, TL2's signal handlers, that are used for emulating non-faulting load instruction on the x86 architecture, catch the resulting segmentation fault and mask the problem. In some rare cases, however, the second free of the object corrupted the heap and subsequent calls to `malloc` or `free` raised the exception and aborted the program.

A newer version of TL2 for x86 fixed this problem by appropriately handling the read-set overflows. This problem was also not very hard to fix, once it was detected and STMBench7 helped me with detecting it. In this particular case, the library authors discovered and fixed the problem on their own, prior to my report.

3.5.3 Other examples

Since performing the experiments described in this chapter, STMBench7 was used in several instances to test various STM implementations. In those instances, as well as in the cases that are mentioned above, STMBench7 proved to be an invaluable tool for stress testing STMs. For example, in the Velox Project [5] STMBench7 was used to highlight the deficiencies of the proposed STM compilers and HTM solutions. As a result, a simplified version of STMBench7 was produced. This STMBench7 version uses a smaller data structure and simpler operations, which makes it easier to support by TM implementations. Similarly, STMBench7 revealed bugs in the Intel's prototype STM C++ compiler, as discussed in Section 5.5.

3.6 Programming issues

STMBench7 effectively highlights the problems that might be encountered by programmers or compiler vendors when they decide to start using STM in production. The identified problems

fall into three main categories: lack of support for external libraries, lack of complete support for object-oriented language features, and difficulties with debugging and testing. For a discussion of several other programming issues with library based STMs, see [23].

3.6.1 External libraries

STMBench7 uses standard library character strings and basic container classes. Even the use of these relatively simple classes was enough to prevent the full utilization of some STMs. For example, both TL2 and TinySTM operate directly at the level of memory words and thus they do not integrate well with external libraries. For that reason, I decided not to implement STMBench7 using the low-level word-based interface directly, as that would require rewriting large parts of the standard library. Instead, I worked around this limitation of word-based STMs by implementing the object-oriented wrapper on top of their interfaces, as described in Section 3.3.3.

A different problem that made the use of standard classes with RSTM more difficult, was RSTM's implementation of the memory allocator interface intended for use in the standard collection classes. The allocator exposed by RSTM replaces standard allocation and deallocation routines with their transactional equivalents, and is thus suitable for a fully transactional implementation of the standard library, which does not exist yet. When a transaction creates an object of a standard, non-transactional class using the allocator, and later aborts, the object's memory gets deallocated twice: once by the object's destructor and once by the rollback of the transaction. This results in heap corruptions and, effectively, prevents the programmer from using objects of standard library classes. To use the standard library, I replaced transactional versions of memory allocation routines in RSTM's version of STL allocator with non-transactional ones. More generally, this leads to a conclusion that STMs which provide support for non-transactional versions of external classes need to provide both transactional and non-transactional memory allocation mechanisms.

3.6.2 Object-oriented features

I also encountered several less severe programming issues, which can make programming with STM more difficult. Most of them were rather easy to fix, but required a deeper knowledge of the STM's internal implementation details. An example of these issues is the lack of support for polymorphism in RSTM's smart pointers. None of the micro-benchmarks revealed this issue, as they use simple class hierarchies which do not take advantage of object polymorphism. For example, the most common micro-benchmarks use classes only to represent the container types, such as trees and lists, and their internal nodes. On the other hand STMBench7 uses inheritance and polymorphism to represent assemblies and their two specializations: base and complex assemblies. As each complex assembly references several child assemblies, which can be either base or complex, it is natural to use polymorphic functions when processing

the children of a complex assembly. Therefore, it is important to use the STM that supports polymorphic transactional objects.

Another issue was related to the fact that TinySTM's memory manager does not invoke destructors when objects are deleted. Instead it simply deallocates the corresponding memory. This results in problems with C++ code that relies on destructors. The simplest solution in programs that only use objects, like STMBench7, is to replace the calls to `free` with calls to `delete`, which invoke the appropriate destructors. However, a more complete solution is needed for programs that need to allocate and deallocate both objects and opaque memory blocks.

3.6.3 Non-faulting loads

My experience with TL2 and TinySTM showed that using signal handlers to emulate the non-faulting load instruction, although correct if used appropriately, can cause significant problems while programming. The main problem with these signal handlers is that they mask actual errors in the program that are results of incorrectly written code, which makes it much more difficult to debug the code during development. I often found it hard to distinguish between a genuine programming error and the error which is the result of inconsistencies that the signal handlers are supposed to mask. Therefore, I believe that simulating non-faulting load instruction in such a way should be avoided. Instead, STMs should either mask the errors in a different way, for example inside a virtual machine for managed languages where there is more control over the execution, or completely eliminate the need for such instructions by, for instance, providing a safe memory manager.

3.7 Summary

To summarize, this chapter presents the conclusions from implementation and experimentation with STMBench7 and several state-of-the-art STMs. In short, STMBench7 can, very effectively, be used to compare the performance of different *policies* as opposed to comparing *mechanisms*. This is in contrast to most commonly used micro-benchmarks. By using STMBench7, I concluded that the combination of the conflict detection technique and the contention manager has the biggest influence on the performance of STM with large-scale workloads. Although other issues, like the choice between a locking and a non-blocking implementation, also influence the overall performance, their impact is not as significant. I conclude that the best conflict detection techniques detect write-write conflicts early and read-write conflicts late. Such a conflict detection scheme is best coupled with Greedy contention manager which achieves the best performance with eager conflict detection approaches.

Furthermore, STMBench7 proved to be a great test of whether an STM is correct, and, more importantly, whether it is truly dynamic and unbounded in practice. Strangely enough, all

STMs I used turned out to crash, for various reasons that were mostly related to memory management.

Finally, STMBench7 can also be used to assess the costs of adopting STM solutions in terms of usability and required changes to external libraries, which might be a very valuable information when choosing an STM for production use. The presented experiments also demonstrated that proper compiler and runtime support, in particular the support for standard libraries, are needed for STM to be mode widely used.

4 SwissTM

In this chapter I propose SwissTM, a new STM I designed based on the conclusions regarding performance of STMs with large-scale workloads from the previous chapter. However, I do not focus solely on large transactions: whereas the performance with large transactions is important, in many cases STM will be used in programs that also use short and simple transactions. My motivation is, therefore, to explore the ability of software mechanisms to effectively support mixed workloads consisting of small and large transactions, as well as possibly complex data structures.

This chapter describes the design and the implementation of SwissTM in detail. It also presents the evaluation of SwissTM using STMBench7, and several other benchmarks: STAMP [17], Lee-TM [8], and standard red-black tree micro-benchmark, used for example in [28, 43, 63]. The evaluation compares the performance of SwissTM to the STMs from the previous chapter: RSTM [77], TL2 [28], and TinySTM [43]. This chapter also "dissects" SwissTM by evaluating the individual impact of several design and implementation choices on its performance, providing valuable insight into when to use each of them in alternative STM designs. Finally, I also describe two extensions to the base SwissTM algorithm: integration with standard STM compilers and support for the privatization idiom.

4.1 Overview

I first briefly summarize the most important findings regarding STM performance from the previous chapter:

1. The lazy acquire scheme, used in, for example, TL2 [28], can indeed be effective for short transactions, but might waste significant work with longer transactions that eventually abort due to write-write conflicts. This is because write-write conflicts, which usually[1] lead to transaction aborts are detected too late.

[1] Pure write-write conflicts do not necessarily lead to transaction aborts in STMs that lazily acquire objects, but are very rare, as most transactions read memory locations before updating them.

2. The eager acquire scheme, used in, for example, TinySTM [43], McRT-STM [99], and Bartok-STM [57], immediately aborts a transaction that tries to read a memory location locked for writing by another transaction. Hence, read-write conflicts, which can often be handled without aborts, are detected very early and are resolved by aborting readers. Long transactions that update memory locations commonly read by other transactions might thus end up blocking many other transactions, possibly for a long time, thus slowing down the system overall.

3. The Timid contention management scheme, often used by lock-based STMs, such as TL2 and TinySTM, which aborts transactions immediately upon a conflict, works well with short transactions. Contention managers such as Greedy [50] or Serializer [77] are more appropriate for large transactions, but are hardly ever used due to the overheads they impose on short ones.

It is appealing but challenging to come up with strategies that account both for long transactions and complex workloads, and for short transactions and simple data structures, which are likely to coexist in real applications. Starting from the conclusions from the experiments with large-scale workloads presented in the previous chapter, I build a lock- and word-based SwissTM which uses the following policies:

- A conflict detection scheme that detects (1) write-write conflicts eagerly, in order to prevent transactions that are doomed to abort from running and wasting resources, and (2) read-write conflicts late, in order to optimistically allow more parallelism between transactions. In short, transactions eagerly acquire objects for writing, which helps detect write-write conflicts as soon as they appear. This also avoids wasting work of transactions that are already doomed to abort due to a write-write conflict. By using invisible reads and allowing transactions to read objects acquired for writing by other transactions, SwissTM detects read-write conflicts late, thus increasing inter-transaction parallelism. A time-based scheme [28, 43] is used to reduce the cost of transaction validation with invisible reads.

- The Two-phase contention manager that incurs negligible overheads on read-only and short read-write transactions while favoring the progress of transactions that have performed a significant number of updates. Basically, transactions that are short or read-only use the simple but inexpensive Timid contention management scheme, aborting on first encountered conflict. Transactions that are more complex switch dynamically to the Greedy policy that involves more overhead but favors complex transactions. Additionally, transactions that abort due to write-write conflicts back-off for a period proportional to the number of their successive aborts, thus reducing contention on memory hot spots.

I compare performance of SwissTM to RSTM [77], TL2 [28], and TinySTM [43] for the reasons that are largely the same as for using them in the previous chapter. These STMs constitute the

STM design choices			
Acquire	*Reads*	*CM*	*Effectiveness*
lazy	invisible	any	+
eager	visible	any	+
eager	invisible	Polka	+
eager	invisible	Timid or Greedy	++
mixed	invisible	Timid or Greedy	+++
mixed	invisible	Two-phase	++++

Table 4.1: A comparison of selected STM designs for mixed workloads.

state-of-the-art performance-wise, among the publicly available library-based STMs. Furthermore, just like SwissTM, they can be used to manually instrument concurrent applications with STM read and write calls. My goal is to evaluate the performance of the core STM algorithm, not the efficiency of the higher layers such as STM compilers. For this reason, I did not use, for instance, McRT-STM [85, 99] as it does not expose such a low-level API to programmers. Evaluating STM compilers, which naturally introduce additional overheads above the low-level STM interface [19, 38, 121], is largely an orthogonal issue. Also, the selected STMs represent a wide spectrum of known TM design choices including: obstruction-free and lock-based progress, direct and deferred updates, visible and invisible reads, and word- and object-level access granularities. They also allow for experiments with a variety of contention management strategies, from simply aborting a transaction on a conflict, through exponential back-off, to advanced contention managers like Greedy [50], Serializer [77], or Polka [100]. This makes them a good choice both for a comprehensive performance evaluation of SwissTM as well as for drawing general conclusions about STM performance. Table 4.1 summarizes the effectiveness of several STM designs to support mixed workloads, based on the performed experiments.

The evaluation is based on benchmarks that cover a large part of the complexity space. STMBench7 [52] is used to represent workloads with non-uniform data structures of significant size, and a mix of operations of various lengths and data access patterns. The evaluation also uses Lee-TM [8], a benchmark with large but regular transactions, and STAMP [17], a collection of realistic medium-scale workloads. Finally, a red-black tree micro-benchmark is used to evaluate low-level overheads of read and write calls on a benchmark that involves very short and simple transactions.

SwissTM outperforms all other used STMs on most of the considered benchmarks and matches their performance on the rest. For example, on the read-dominated workload of STMBench7, with 90% of read-only operations, SwissTM outperforms the other STMs by more than 55%, and on the read-write workload, with 60% of read-only operations, by more than 40%. Also, SwissTM exhibits better scalability than the other STMs, especially for read-dominated and read-write workloads of STMBench7. Similarly, SwissTM achieves good performance on other used workloads.

I also evaluated the impact of individual design choices on the performance of SwissTM. For example, I demonstrate that using the mixed eager-lazy conflict detection instead of the pure eager scheme helps significantly in face of long-lasting read-write conflicts, by comparing SwissTM and TinySTM on a modified version of Lee-TM benchmark. Two-phase contention manager is also evaluated in detail: I demonstrate that it is a better choice than Polka or Timid on the large-scale STMBench7 benchmark, and that it outperforms pure Greedy scheme on the small-scale red-black tree micro-benchmark. The effectiveness of the selected linear back-off scheme is also illustrated, using the *intruder* benchmark from STAMP as an example. From an implementation perspective, I present a detailed evaluation of impact of locking granularity on SwissTM performance. Word-based STM implementations typically use either word-level locking, like TL2 and TinySTM, or cache-line level locking, like McRT-STM. The sensitivity analysis I performed shows that the locking granularity of four words outperforms both word and cache-line granularities by 4% and 5% respectively averaged across all considered benchmarks.

4.2 Design and implementation

SwissTM is a lock-based STM that uses invisible reads. It relies on the shared time-base to optimize reads-set validations, as described in Section 3.2 and employed by TL2 and TinySTM. It uses eager write-write and lazy read-write conflict detection, as well as the Two-phase contention manager with random linear back-off. SwissTM is word-based, enabling transactional access to arbitrary memory words. It uses deferred updates and, hence, a redo-logging scheme, in part to support the lazy detection of read-write conflicts.

4.2.1 Programming model

When programming with SwissTM, programmers have to manually replace all memory references to shared data from inside transactions with STM calls for reading and writing memory words. As discussed, this programming model can be improved by using an STM compiler, such as [45, 57, 85]. While the compiler instrumentation can degrade performance due to over-instrumentation [19, 38, 121] and possibly even change the characteristics of the workload slightly by, for example, changing numbers and ratio of transactional reads and writes, the compiler instrumentation remains a largely orthogonal issue to the performance of an STM library. Discussion of how to integrate SwissTM with standard STM compilers is deferred to Section 4.5.1. Section 5.5 evaluates the performance impact of using an STM compiler with SwissTM across a range of workloads.

Similarly to most other STM libraries, SwissTM guarantees opacity [51]. SwissTM is a weakly atomic STM, thus not providing any guarantees for code that accesses the same data from both inside and outside of transactions. Base SwissTM variant is not privatization safe [106], which might make programming with SwissTM slightly more difficult in certain cases, but it did not affect the experiments, as none of the benchmarks requires a privatization-safe

STM. Discussion of how to extend SwissTM to support the privatization idiom, using standard techniques, is deferred to Section 4.5.2. Section 5.6 evaluates the performance impact of guaranteeing privatization safety in SwissTM across a range of workloads.

Other three STMs used in the experiments provide the same semantical guarantees as SwissTM. I believe that strengthening the guarantees would have a similar performance impact on them as on SwissTM, thus making the performance comparison fair.

4.2.2 Algorithm

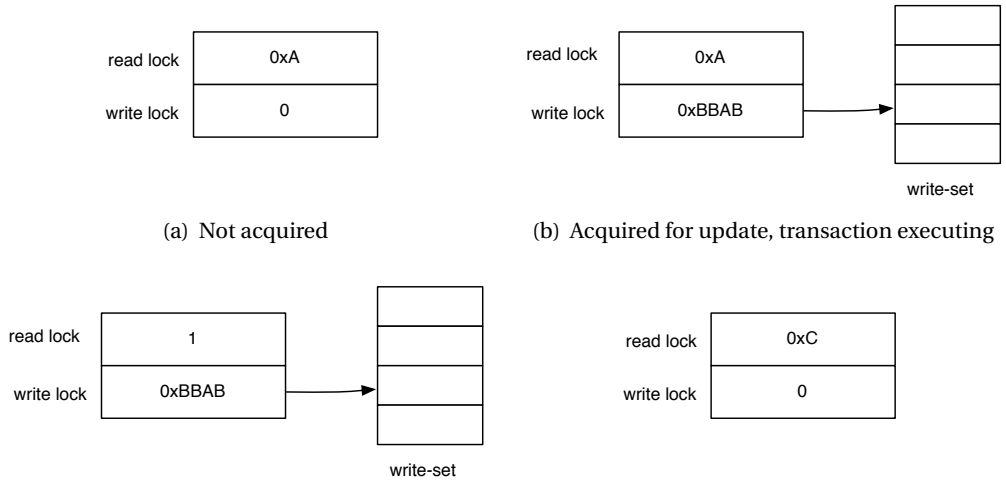
Pseudo-code. The pseudo-code of SwissTM uses a C/C++-like syntax, as this makes it easier to distinguish between pointer and value variables. It assumes that atomic *fetch-and-increment* and *compare-and-swap* instructions are supported by the CPU, which is the case for most modern CPUs. For simplicity, the code also assumes a sequentially consistent memory model and ignores version overflows. The pseudo-code uses a data structure for maintaining read- and write-sets, which supports operations typical of a collection class. I discuss these assumptions and their implications on the implementation in more detail in Section 4.2.4.

For clarity, the pseudo-code of SwissTM is split into several figures: used types and global variables are defined in Figure 4.2, the main part of SwissTM algorithm is presented in Figure 4.3, and the contention manager pseudo-code is given in Figure 4.5.

Ownership records. SwissTM uses meta-data organized in *ownership records*, or *orecs* for short, to keep track of accesses to memory locations. To simplify the presentation, the pseudo-code assumes that each ownership record corresponds to a single memory location. The details of storing and accessing the ownership records are discussed in Section 4.2.4, together with other implementation issues.

The ownership record consists of a read and a write lock. Both of these locks are acquired by writers: the read lock is used to detect read-write conflicts and the write lock is used to detect write-write conflicts. When a transaction acquires a particular write lock, it stores to it a pointer to the corresponding write-set entry. This facilitates fast read-after-write and write-after-write checks. Write locks are set to zero when released. When a transaction acquires a particular read lock it sets its value to 1. Otherwise the read locks are set to the current *orec* version, shifted to the left by one. This scheme enables transactions to distinguish between an acquired and a free read lock by simply checking whether its least significant bit is set or not.

By using two locks instead of one, as is usually done in other STMs, SwissTM is able to detect different types of conflicts at different points in the execution: write locks are acquired at the time of write, to eagerly detect write-write conflicts, and read locks are acquired at commit time, to lazily detect read-write conflicts. With this approach, a read lock is always acquired by the transaction that already holds the corresponding write lock. Consequently, transactions



(a) Not acquired

(b) Acquired for update, transaction executing

(c) Acquired for update, transaction committing

(d) Not acquired, after commit

Figure 4.1: Different states of an ownership record.

acquire read locks using an ordinary write, in contrast to acquiring write locks, which require use of atomic *compare-and-swap*.

Figure 4.1 illustrates different states of an *orec*, when the location it corresponds to is updated by a transaction. The *orec* starts unlocked with version equal to 5, as depicted in Figure 4.1(a). Thus the read lock holds value *0xA* which corresponds to the *orec* version shifted to the left by one. When a transaction acquires the write lock, it stores a pointer to its write-set entry, as in Figure 4.1(b). The read lock is not updated at this time, so the concurrent readers can proceed with reading the location without conflicting with the writer. Figure 4.1(c) depicts what happens at commit time, when the writer transaction acquires the read lock too. In this state, both the read and the write lock are held by the same transaction, which now has exclusive access to the corresponding memory location. After commit, both locks are released, with the read lock updated to contain the new *orec* version, equal to 6 in the example, shifted to the left by one position, as in Figure 4.1(d).

Data structures. Pseudo-code in Figure 4.2 defines types and global data used by SwissTM. It includes the declaration of an architecture-specific word type, which is the granularity at which SwissTM detects conflicts and logs updates. SwissTM relies on the system-specific long-jump buffer, declared in the figure, to handle transaction restarts. The pseudo-code also declares the log and log iterator types, provided by the implementation, and the ownership record, consisting of a read and a write lock. The transaction descriptor maintains book-keeping data local to a transaction. Apart from the long-jump buffer used for restarts, it maintains the read-set validity timestamp and read- and write-sets. It also maintains several variables used for contention management, which are discussed below.

```
1  // architecture-specific word type
2  typedef word_t;
3  // system-defined long jump-buffer
4  typedef jmpbuf_t;
5  // implementation-specific log type
6  struct log_t;
7  struct iter_t;
8  // lock type
9  struct TxOrec {
10   word_t write_lock;
11   word_t read_lock;
12 };
13
14 // transaction-local descriptor
15 struct TxDescriptor {
16   jmpbuf_t jmpbuf;
17   word_t valid_ts;
18   log_t read_set;
19   log_t write_set;
20   // contention manager
21   word_t cm_ts;
22   int succ_aborts;
23   bool aborted;
24 };
25 // shared time-base
26 word_t commit_ts;
```

Figure 4.2: SwissTM pseudo-code (types and shared data).

Global variables. The only global variable SwissTM uses is the shared time-base, denoted as `commit_ts`, which is used to speed-up read-set validations, as discussed in Section 3.2 and below.

Transaction start. The code that uses SwissTM sets a long-jump buffer to the instruction that invokes `TxStart`. The long-jump buffer is then passed to the `TxStart` call as an argument and is stored in the descriptor (Figure 4.3, line 3). When a transaction aborts, it restarts itself by executing a long-jump to the provided long-jump buffer, effectively transferring execution to the start of the transaction (line 101). At its start, the transaction also reads the shared time-base `commit_ts` and stores its value in the descriptor as its validity timestamp (line 4).

Transactional read. To read a memory location, a transaction performs the following steps:

1. It maps the location to the corresponding *orec* (lines 10–12).

2. It checks whether it previously acquired the corresponding write lock, in which case the read needs to return the value of transaction's last write to the location (line 14). If it did, the read immediately returns the location's new value stored in the write-set (line 15), ensuring that read-after-write accesses are handled correctly.

3. The transaction next reads consistent values of the read lock and the memory location (lines 17–28). To do so, it reads the read lock, then the location, and then the read lock again. These steps are repeated until the two values of the read lock match, and the lock is not held by any transaction. Spinning here is safe: the owner of the lock is in its commit phase and, hence, it does not try to acquire any additional write locks, which eliminates the possibility of a deadlock with the transaction that is performing the read. As the lock owner is already in its commit phase, it will soon release the lock, so the wait at this point is expected to be short. The alternative to waiting is to abort the current transaction or the owner of the lock, but neither is beneficial: the owner will soon

```
1  void TxStart(TxDescriptor *tx,
2               jmpbuf_t jmpbuf) {
3    tx->jmpbuf = jmpbuf;
4    tx->valid_ts = commit_ts;
5    cm_on_start(tx);
6  }
7  word_t TxReadWord(TxDescriptor *tx,
8                    word_t *addr) {
9    // map address to meta-data
10   TxOrec *lock = map_addr_to_lock(addr);
11   word_t *wlock = &lock->write_lock;
12   word_t *rlock = &lock->read_lock;
13   // check for read-after-write
14   if(is_locked_by_me(tx, wlock))
15     return get_write_value(wlock, addr);
16   // read version and value consistently
17   word_t version = *rlock;
18   while(true) {
19     if(is_locked(version)) {
20       version = *rlock;
21       continue;
22     }
23     word_t value = *addr;
24     word_t version2 = *rlock;
25     if(version == version2)
26       break;
27     version2 = version;
28   }
29   // validate read set if needed
30   add_to_read_set(tx, rlock, version);
31   if(version > tx->valid_tx)
32     if(!extend(tx))
33       rollback(tx);
34   return value;
35 }
36 void TxWriteWord(TxDescriptor *tx,
37                  word_t *addr,
38                  word_t val) {
39   // map address to meta-data
40   TxOrec *lock = map_addr_to_lock(addr);
41   word_t *wlock = &lock->write_lock;
42   word_t *rlock = &lock->read_lock;
43   // check for write-after-write
44   if(is_locked_by_me(tx, wlock)) {
45     store_write_value(wlock, addr, val);
46     return;
47   }
48   // acquire the write lock
49   while(true) {
50     word_t wl = *wlock;
51     if(is_locked(wlval) {
52       if(cm_should_abort(tx, wlval))
53         rollback(tx);
54       continue;
55     }
56     word_t entry = add_to_write_set(
57       tx, wlock, addr);
58     if(compare_and_swap(
59         wlock, UNLOCKED, entry))
60       break;
61     return_to_write_set(tx, entry);
62   }
63   // validate read set if needed
64   if(*rlock > tx->valid_ts)
65     if(!extend(tx))
66       rollback(tx);
67   cm_on_write(tx);
68 }
69 void TxCommit(TxDescriptor *tx) {
70   // check if read-only
71   if(is_read_only(tx)) return;
72   // lock read locks
73   for(iter_t *it in tx->write_set) {
74     *it->read_lock = LOCKED;
75   }
76   // get commit timestamp
77   word_t ts = fetch_and_increment(
78       &commits_ts);
79   // validate read set
80   if(ts > tx->valid_ts + 1) {
81     if(!validate_commit(tx)) {
82       for(iter_t *it in tx->write_set) {
83          *it->read_lock = it->version;
84       }
85       rollback(tx);
86     }
87   }
88   // copy values to memory
89   for(iter_t *it in tx->write_set) {
90     *it->address = it->value;
91     *it->read_lock = (ts << 1);
92     *it->write_lock = UNLOCKED;
93   }
94   cm_on_commit(tx);
95 }
96 void rollback(TxDescriptor *tx) {
97   for(iter_t *it in tx->write_set) {
98     *it->write_lock = UNLOCKED;
99   }
100  cm_on_rollback(tx);
101  longjmp(tx->jmpbuf);
102 }
103 bool extend(TxDescriptor *tx) {
104   word_t ts = commit_ts;
105   if(validate(tx)) {
106     tx->valid_ts = ts;
107     return true;
108   }
109   return false;
110 }
111 bool validate(TxDescriptor *tx) {
112   for(iter_t *it in tx->read_set) {
113     if(it->version != *it->read_lock) {
114       return false;
115     }
116   }
117   return true;
118 }
119 bool validate_commit(TxDescriptor *tx) {
120   for(iter_t *it in tx->read_set) {
121     word_t rlval = *it->read_lock
122     if(it->version != rlval)
123       if(!is_locked_by_me(tx, rlval) {
124         return false;
125       }
126   }
127   return true;
128 }
```

Figure 4.3: SwissTM pseudo-code (base algorithm).

commit, so the current transaction needs to wait only for a short while, and aborting either transaction would result in an unnecessary waste of work they performed.

4. Then, the transaction validates the read set (lines 30–33). Before validation, it stores the address of the read lock and its version into the read set, to ensure that the *orec* does not change during the validation. This ensures that races between the ongoing read and concurrent updates to the same location are avoided. To validate the read-set, the transaction first compares *orec*'s version to the transaction's validity timestamp. If the *orec*'s version is lower or equal to the timestamp, the read-set is still consistent and the full validation is not necessary. If it is higher, then the transaction tries to extend validity of the read-set (lines 103–110).

   To do so, the transaction first reads the current value of the shared time-base counter `commit_ts` and then checks whether any of the *orecs* in its read-set have been updated since first accessed by the transaction (lines 111–118). If all *orecs* are still the same, the validation is successful and the read-set validity timestamp can be updated to the value of `commit_ts` observed before the validation. By first reading `commit_ts` and then validating the read-set, the transaction ensures that the read-set was valid at the time it read `commit_ts`, thus making it safe to adopt the `commit_ts`'s value as its new validity timestamp. Whereas the read-set extension takes work proportional to the read-set's size, it is not performed very often because the *orec*'s version is usually lower than transaction's validity timestamp, so it does not impact performance significantly. In cases when the version of the newly accessed *orec* is higher than the validity timestamp, the transaction has two choices: either to abort immediately, or to try to extend the read-set validity. The latter option tries to reduce the amount of work wasted by aborts, and this is why it is used in SwissTM.

5. Finally, if the read-set is valid, the value read from the location is returned to the caller.

Transactional write. To update a memory location, a transaction performs the following steps:

1. As when reading, it first maps the location to the corresponding *orec* (lines 40–42).

2. The transaction then checks whether it has previously acquired the write lock in the *orec*. If it has, the corresponding value in the write-set is updated, ensuring that write-after-write accesses are handled correctly, and the write call returns (lines 44–47).

3. Next, the transaction attempts to acquire the write lock (lines 49–62). To do so, it first checks whether the lock is held by another transaction. If it is, the transaction invokes the contention manager (line 52) to check whether it should abort. In case it is allowed to keep executing, the owner of the lock will be signaled to abort. However, the lock owner will not immediately detect that it has been aborted and even once it does, it will take some time for it to release the lock during rollback. For that reason, when the

transaction wins the contention management, it simply keeps spinning until the lock is released.

After the transaction observes that the lock has been released, it prepares a new write-set entry and uses atomic *compare-and-swap* instruction to acquire the lock by storing the address of the entry into the lock (lines 56–58). If the acquire attempt fails, the write-set entry is returned to the write-set, and the transaction performs the above steps again in a repeated attempt to acquire the lock. Acquiring the lock fails only if another transaction manages to acquire it first, between the time of reading the write lock and attempting the *compare-and-swap*. When the acquire fails, the transaction simply re-reads the write lock and performs contention management with the new owner of the lock, spinning until either the acquire attempt succeeds, or the contention manager decides to abort the transaction. Spinning here is safe because the contention manager ensures that transactions do not deadlock.

4. The transaction next validates its read-set (lines 64–66), using the version of the *orec* it has acquired. The validation proceeds as with the read and if it succeeds, the write call returns. The reason for validating the read-set even when writing might not be obvious at first. It is described below why the validation is indeed necessary, after the description of transaction commit.

Transaction commit. A read-only transaction commits immediately without any checks (line 71), as its read-set is guaranteed to have been consistent at the time of the last read.

A read-write transaction performs the following steps:

1. It first acquires read locks of all locations in its write-set (lines 73–75). The read locks can be acquired using a simple store, as the transaction already holds the corresponding write locks.

2. Next, the transaction obtains a unique commit timestamp by atomically incrementing the shared time-base counter `commit_ts` (line 77), using *fetch-and-increment* instruction. This value is used as the new version for all updated locations. The time when *fetch-and-increment* is performed represents the unique point at which the transaction is serialized in the global ordering of transactions.

3. Transaction then validates its read-set (lines 80 and 81). The validation is not necessary if transaction's commit timestamp is higher than its validity timestamp by one, which means that no transaction committed since the last read-set validation. The commit-time validation is similar to the validation performed during transaction's execution (lines 119–128). The only difference is that it also checks whether the read lock has been acquired by the current transaction to correctly handle locations that are first read, and then updated by the same transaction. If the validation fails, the transaction first

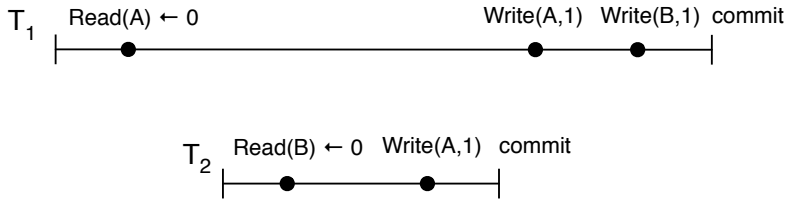


Figure 4.4: A non-serializable execution permitted if there is no validation on write.

releases all read locks and then performs a rollback as usual (lines 82–86). If the read-set is valid, the commit succeeds.

4. The transaction then copies the values from its redo-log to memory locations, and then releases the read and the write locks, in that order (lines 89–93). The transaction releases the read locks by setting them to the commit timestamp shifted left by one position, and write locks by setting them to zero, denoted as special value `UNLOCKED` in the pseudo-code.

Validation on write. The reason for validating the read-set when writing should be clearer now: the commit-time validation simply ignores the read locks that have been acquired by the committing transaction, assuming that the read-set validity was ensured before the corresponding read lock was acquired (line 123). Figure 4.4 depicts a classical example of a non-serializable execution that would be permitted by SwissTM if the read-set validation was not performed when writing. In the figure, transaction T_1 reads variable A, getting A's old value 0, and then writes 1 to variable B. Concurrently, T_2 reads B, getting B's old value 0, and then writes 1 to A. Clearly, there is no serializable execution of transactions T_1 and T_2, so transaction T_1 has to be aborted. However, if transaction T_1 also writes to A after T_2 commits, as in the figure, and does not validate its read-set at that point, the commit validation does not detect the conflict with T_2, and T_1 is allowed to commit. This is clearly incorrect, as no serializable execution of T_1 and T_2 exists. The read-set validations performed on writes are necessary to eliminate these and similar inconsistencies.

There are several alternatives to validating the read-set on every write:

1. The validation is necessary only if the write is preceded by the read of the same location. Transactions could, therefore, search their read-sets and perform the validation only if they actually previously read the location being updated.

2. Transactions could validate their read-sets before acquiring the read-locks during the commit (line 73). Doing so would detect potential read-set inconsistencies at commit time. Note that the validation after acquiring the read locks is also necessary in this case.

```
1  // shared counter for second phase
2  word_t greedy_ts;
3
4  void cm_on_start(TxDescriptor *tx) {
5    if(tx->succ_aborts == 0) {
6      tx->cm_ts = INF;
7    }
8  }
9  void cm_on_write(TxDescriptor *tx) {
10   if(tx->cm_ts == INF)
11     if(size(tx->write_set) == CM_THRS) {
12       tx->cm_ts = fetch_and_increment(
13                     &greedy_ts);
14     }
15   }
16 }
17 void cm_on_commit(TxDescriptor *tx) {
18   tx->succ_aborts = 0;
19   tx->aborted = false;
20 }
21 void cm_on_rollback(TxDescriptor *tx) {
22   tx->succ_aborts++;
23   wait_random(tx->succ_aborts);
24 }
25 bool cm_should_abort(TxDescriptor *tx,
26                      word_t *wlock) {
27   // check if aborted
28   if(aborted)
29     return true;
30   // first phase
31   if(tx->cm_ts == INF)
32     return true;
33   // second phase
34   TxDescriptor *other = owner(wlock);
35   if(other->cm_ts == INF) {
36     other->aborted = true;
37     return false;
38   }
39   if(other->cm_ts > tx->cm_ts) {
40     other->aborted = true;
41     return false;
42   }
43   return true;
44 }
```

Figure 4.5: SwissTM pseudo-code (contention manager).

3. Transactions could store the version of the read-locks into the write-set entries before acquiring them. The validation at commit time would then check whether the version in the read-set matches the version in the write-set.

Each of these approaches has their advantages and limitations, but the choice does not impact the performance significantly. I opted for validation on write, because it (1) detects the existing conflicts early, thus reducing the amount of wasted work by transactions that are doomed to abort, and (2) does not require read-set traversals during writes.

Transaction rollback. A transaction is rolled back by releasing the write locks it has acquired and jumping to the jump buffer provided by the client code at transaction start (lines 96–102). There is no need to release the read locks, as they are held only during transaction commit and are released explicitly if the transaction aborts after acquiring them (lines 82–84).

Contention manager. Figure 4.5 contains the pseudo-code of the Two-phase contention manager. The main part of the SwissTM algorithm communicates occurrence of certain events in the transaction's execution to the contention manager by invoking the appropriate callback functions provided by the contention manager. The contention manager defines callbacks for transaction's start, rollback, and commit, and transactional writes. In addition, it also provides a function that transactions invoke to resolve write-write conflicts.

The contention manager uses several fields in the transaction descriptor: the timestamp used to order long update transactions in the second phase of contention management, the number of successive aborts used to calculate back-off interval on restarts, and the indication

of whether the transaction has been aborted or not. The contention management timestamp is also used to indicate which phase of contention management the transaction is in: it is set to a special value `INF` in the first phase and to the timestamp value in the second. The contention manager uses global variable `greedy_ts` to generate contention management timestamps.

On each of the events in transaction execution the contention manager performs the following actions:

- When a transaction starts for the first time, its timestamp is set to `INF`, to have transactions start in the first phase of contention management (line 6). This timestamp is not modified on restarts, to keep the assigned timestamp value across restarts.
- When a transaction in the first contention management phase performs a write, the contention manager checks whether the number of writes has become higher than a predefined threshold used to switch between contention management phases.[2] If it has, the transaction moves to the second phase and obtains a unique contention management timestamp by atomically incrementing the shared counter `greedy_ts` (lines 10–15).
- On transaction commit, the number of successive aborts is reset to 0, to indicate that the next call to `TxStart` is not a restart, but a start of a new transaction (line 18).
- On transaction rollback, the number of successive aborts is incremented by one and the transaction spins for a short while to reduce contention (lines 22 and 23). To calculate the back-off interval, the transaction randomly generates a number lower than the maximum back-off interval, which is linearly proportional to the number of successive aborts. The transaction restarts by performing a long jump, as described above, only after performing the back-off.

When a transaction detects a write-write conflict, it invokes `cm_should_abort` function of the contention manager. The return value of this function indicates whether the transaction should abort or not. If the function returns `false`, it always first sets the `aborted` flag in the descriptor of the current lock owner, thus ensuring that two transactions never deadlock (lines 36, 37 and 40, 41). A transaction checks whether the `aborted` flag is set, meaning it should abort, only when it encounters a conflict itself and tries to resolve it by invoking `cm_should_abort` (lines 28 and 29). Thus, transactions are allowed to continue executing as long as they do not encounter a conflict, even if another transaction is waiting for them to release a lock. This approach eliminates frequent checks of the `aborted` flag and guarantees that transactions which do not get blocked by other transactions commit. Once a conflict is detected, the transaction first checks if it has already been aborted by some other transaction before checking whether it has priority over the lock owner.

[2] The threshold is set to 10 in the implementation as this value results in good performance.

To decide whether to abort itself or the lock owner, the transaction performs the following steps (lines 31–43). In the first phase of the contention management, the transaction immediately aborts (lines 31 and 32). In the second phase, the transaction first checks which phase of contention management the owner of the lock is in. If the owner is in the first phase, it is signaled to abort and the transaction keeps executing (lines 35–38). If the owner is in the second phase as well, then the contention management timestamps are compared and the transaction with the lower timestamp wins (lines 39–42). In this way, the contention manager prioritizes longer transactions that have performed more work, over shorter ones.

4.2.3 Correctness argument

I did not formally prove correctness of SwissTM. Instead, I rely on intuitive reasoning and extensive testing to ensure that the algorithm is indeed correct. Next, I present the intuitive arguments for SwissTM's correctness.

Atomic updates. Transactions acquire both read and write locks before copying the values to memory. Thus, no other transaction can observe partial results of a commit: if transaction T_1 updates variables A and B, other transactions can observe either old or new values of both variables. Similarly, the updates of two transactions cannot be mixed: if transactions T_1 and T_2 both update variables A and B, with T_1 writing a_1 and b_1 and T_2 writing a_2 and b_2 to them, then transaction T_3, that starts after T_1 and T_2 commit, can observe A and B as either a_1 and b_1 or a_2 and b_2, not as some mix of values written by T_1 and T_2. This means that updates performed by a transaction appear to be atomic to all other transactions.

Consistent reads. When transaction T starts, it reads the current value v_1 of the shared time-base. When reading a memory location, if the version of the location is lower or equal to v_1, that means that the location has not been updated since T started. Thus, it belongs to the memory snapshot that would be taken at the time of T's start, meaning that the read-set is consistent.

It can happen that T gets the commit timestamp of a currently committing transaction T_c, as its validity timestamp, and that it tries to read a memory location that T_c has still not finished updating. In this case, however, the read lock of the location is still held by T_c, as transactions acquire all read locks before acquiring the commit timestamp and release them only after updating the memory locations protected by the locks. The read locks prevent T from reading a mix of old and new values of locations updated by transaction T_c, maintaining the consistency of T's read-set in this case as well.

When transaction T extends validity of its read-set from v_1 to v_2 it ensures that all locations in the read-set have remained the same since its start by performing a full validation of the read-set. If the locations in T's read-set have not changed since it started, then T can continue

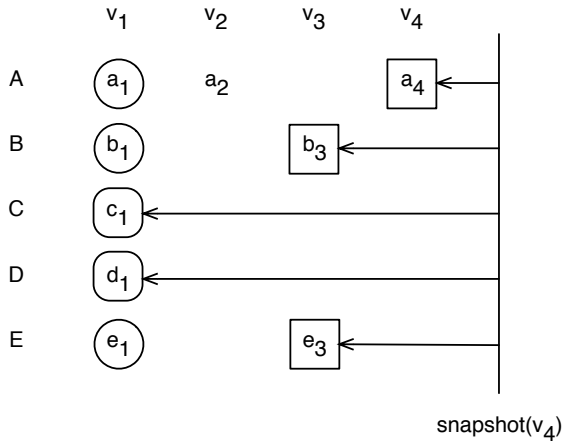


Figure 4.6: Example validity timestamp extension.

executing as if it always executed on the snapshot valid at time v_2. To ensure that v_2 and the validation are consistent during the extension, T first reads the shared time-base, getting v_2, and then performs the validation. This means that the read-set might actually be valid at some point after v_2, which also means that it is valid at v_2, so the extension is correct.

Figure 4.6 illustrates why extension of the read-set validity timestamp is correct. It depicts several variables: A, B, C, D, and E and the history of their values as the time passes and transactions commit. Each point in time, depicted as v_1–v_4, represents a commit of one transaction. In the figure, A was updated at times v_2 and v_4, B and E were updated at time v_3, while variables C and D have not been updated at all. To build a snapshot valid at some point in time, one needs to move from that point in time towards left and select the first value encountered for each variable. The figure illustrates building of a snapshot valid at time v_4, which contains variables a_4, b_3, c_1, d_1, and e_3. The figure depicts snapshots at times v_1 and v_4 by representing the values belonging to the snapshot at v_4 as squares, the values belonging to the snapshot at v_1 as circles, and the values belonging to both snapshots as rounded squares. Clearly, if a transaction starts with a validity timestamp v_1 and reads values c_1 and d_1, it can safely extend its validity timestamp to v_4, as c_1 and d_1 belong both to snapshot at v_1 and at v_4. Hence the extension of read-set validity does not violate the consistency of the read-set.

Serializability. Read-only transaction T is consistent at the time of its last read, which is ensured by the read-set validations. Therefore it serializes to the time of the last read, which corresponds to its validity timestamp.

Update transaction T acquires both read and write locks for all locations in its write-set, obtains the commit timestamp, and validates its read-set before committing values to the updated memory locations. Successful read-set validation means that locations in T's read-set belong to the snapshot taken at the time when the commit timestamp is obtained. The writes are serialized to the same time as well, because both read and write locks are acquired before

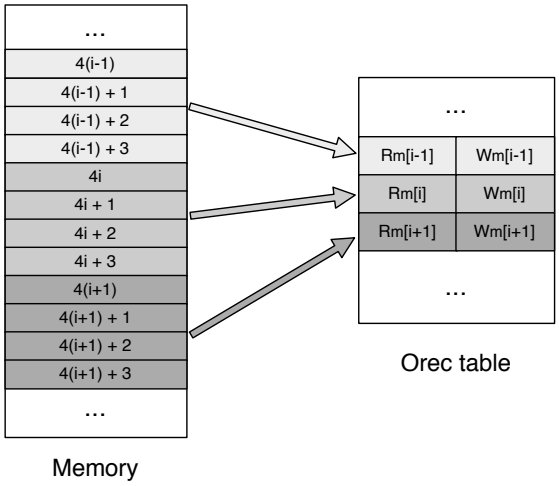


Figure 4.7: SwissTM ownership record table mapping.

obtaining the commit timestamp. Therefore, the whole transaction serializes to the point when the shared time-base is atomically incremented.

4.2.4 Implementation details

This section presents several implementation details that are omitted from the algorithm description for clarity. Most importantly, it describes where the *orecs* are stored, how they are mapped to memory locations, and what the implications of the chosen mapping scheme are. It also presents several other implementation details, including the used memory management scheme, the implementation of the efficient log data structure, the implications of using SwissTM on systems with relaxed memory models, and how transactions deal with version overflows.

Orec table. The algorithm description assumes that memory locations are mapped to *orecs* one-to-one. However, to provide such a one-to-one mapping scheme with a completely general programming interface, which allows transactional accesses to any location in memory, requires preallocating a huge number of *orecs*. For that reason, SwissTM, similarly to, for example, TinySTM and TL2, maintains the *orecs* in a separate global *orec* table, depicted in Figure 4.7. On every access, the address of the accessed memory location is first mapped to an entry in the table to obtain the *orec* corresponding to the location. The mapping is performed by shifting the address of the location right by four positions on 32-bit systems, and by five on 64-bit systems, and setting high order bits to zero. I empirically selected the shift size, as explained in Section 4.4. The *orec* table contains 2^{22} entries, so the high order bits are cleared by performing bitwise *AND* operation between the shifted address and $2^{22} - 1$.

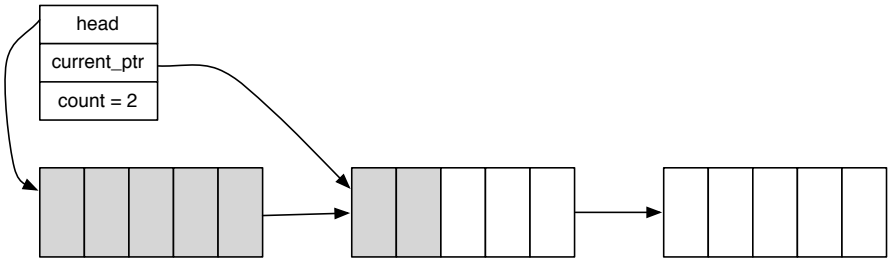


Figure 4.8: Log data structure.

With this scheme, groups of four consecutive memory locations map to one *orec*, as Figure 4.7 illustrates, resulting in a many-to-one mapping between memory locations and *orecs*. Furthermore, such groups of four memory words that are a multiple of 2^{24} words apart also map to the same *orec*. Having multiple memory locations map to the same *orec* can result in *false conflicts*, which occur when two transactions access different memory locations that map to the same *orec* and thus conflict, despite actually accessing different data. False conflicts potentially increase abort rates, thus adversely impacting the performance, but they do not cause noticeable performance impact in practice. They, however, make the implementation of the SwissTM algorithm slightly more complex, as transactions have to deal with the possibility of several words mapping to the same *orec*.

Redo logs. Because multiple memory locations map to the same *orec*, it is not sufficient for write-set entries to store only one address-value pair, as presented in the pseudo-code. Instead, they store a linked-list of address-value pairs, to support updates to multiple memory locations that map to the same *orec*. To amortize costs of memory allocation, each thread keeps a pool of linked-list nodes for storing these address-value pairs, implemented using the log data structure described below. The per-*orec* address-value pair lists need to be traversed whenever a transaction performs a read-after-write or write-after-write access to find the correct address-value pair. Similarly, they are traversed when copying values from redo-log to memory during commits.

Efficient logging. Transactions use an optimized log data structure to maintain their read- and write-sets. The most important requirements for the log data structure is to efficiently support inserts of new elements: these occur frequently, as transactions add new elements to their read- and write-sets during the execution. In particular, it is important to avoid frequent memory allocations when inserting new elements into the log. The log also needs to support efficient traversals because transactions traverse their read-sets, when validating, and their write-sets at commit-time, when acquiring the read locks and when copying updates to the shared memory. It is also important that the log can be emptied quickly, because transactions empty their read- and write-sets when they commit or abort. Furthermore, the log needs to enable the removal of the last inserted element, as the last inserted write-set entry is returned

to the write-set when acquiring of the write lock fails. Finally, it needs to provide a rough estimate of the number of inserted elements, to enable the contention manager to decide whether a transaction should transition to the second phase of contention management or not. The returned number of elements in the log does not need to be accurate, as the contention manager only needs to know if it is higher than the predefined threshold. The log does not need to support any form of element lookup because transactions look up write-set entries using the pointer stored in the write locks and do not look up read-set entries at all.

To efficiently support the required operations, I implemented the log as a doubly linked-list of arrays, illustrated in Figure 4.8. The log is initialized with a single array, and new arrays are added to the list as needed. Each array holds 2048 entries to avoid frequent memory allocations and to support short transactions with a single array. The arrays are not deallocated when the log gets emptied, as future transactions are likely to require similar log sizes. The implementation only maintains the head of the list, the currently used array, and the number of occupied elements in the array. The figure shows a log with three arrays, where each array has five elements and seven elements were inserted into the log. The occupied array elements are depicted as grayed in the figure.

To insert a new element, the counter that tracks the number of occupied elements in the currently used array is incremented. When the current array becomes full, the next array in the list is used. A new array is allocated only when all arrays in the list are fully occupied. In the example log from the figure, that would happen during the sixteenth insert. To remove the last inserted element, the counter is decremented and if the removed element was first in its array, the pointer to the currently used array is set to the previous array. To empty the log, the counter is set to zero and the current array is set to the head of the array list. Log traversals visit all currently used arrays, from the head of the list to the array pointed to by the current array pointer, and traverse the occupied elements in each of the arrays. The described log organization is simple, efficient, and flexible, and it helps SwissTM achieve good performance by reducing logging overheads compared to the standard collection classes.

Version overflows. The presented SwissTM algorithm disregards possibility of overflows of the shared time-base used to generate *orec* versions. If the time-base overflows, the time-based validation does not work correctly: after the overflow, transactions are not able to determine whether *orecs* were updated before or after their start.

On 64-bit architectures the overflows are indeed very unlikely: with version updates occurring on every CPU clock cycle at frequency of 4 GHz, it takes around 2^{30} seconds, or more than 30 years to overflow a 64-bit word. Typical transactions last for at least several thousand cycles, which means that it would take thousands of years to actually overflow a 64-bit shared time-base. However, on 32-bit architectures, the overflows are much more likely. Even if transactions take 10,000 cycles on average, a single CPU at 4GHz would overflow a 32-bit counter in less than an hour. Taking into account that multiple CPUs update the shared

time-base, the overflow could occur much sooner. Therefore, SwissTM needs to correctly handle the overflows of the time-base on 32-bit systems.

To deal with the overflows, each transaction checks its commit timestamp to detect whether an overflow is about to happen just after obtaining it during the commit. If the timestamp is close to the maximum value that can be stored in a memory word, the transaction tries to reset the shared time-base. Transactions do not wait until they observe the maximum value, as that would allow an overflow by several concurrently committing transactions: if one of the transactions increments the shared time-base to its maximum value, the next increment will cause an overflow of the time-base by setting it to zero. To prevent such scenarios, a "safety window" of unused version numbers is reserved. The window has to be larger than the maximum number of concurrently executing threads, which is in the order of hundreds for typical systems.

When a transaction detects that the reset of the time-base is necessary, it aborts itself and signals the other threads that a reset of the time-base is in progress by setting a shared flag. The flag is set atomically using a *compare-and-swap* to prevent multiple transactions from performing the reset at the same time. The transaction that sets the flag waits for all executing transactions to complete, and, after that, it proceeds to reset the time-base and all read locks in the *orec* table by setting them to the initial system version. To ensure that no new transactions start while the reset is in progress, each transaction checks the value of the flag before starting and waits for the ongoing reset to complete. When the reset is done, transactions proceed to execute as usual.

Relaxed memory models. For simplicity, SwissTM pseudo-code assumes sequentially consistent memory model. In reality, however, both the compiler and the hardware reorder instructions to improve performance of the executed code. The reordering could cause problems when instructions need to be executed in the exact order specified by the pseudo-code. For example, reading consistent values of the memory location and the corresponding read lock requires the sequence of three reads that cannot be reordered (lines 17, 23, and 24 in Figure 4.3). Likewise, when releasing the locks during commit, it is essential that values get copied to memory first, the read lock gets released next, and only then the write lock gets released too (lines 90–92). To prevent reordering of instruction in these and similar cases, declared the shared variables as `volatile` and used the necessary memory barriers in the implementation.

Memory manager. SwissTM supports transactional allocation and deallocation of memory blocks. When performing memory management operations transactions log the operations during their execution, and make them permanent if they commit or rollback them if they abort.

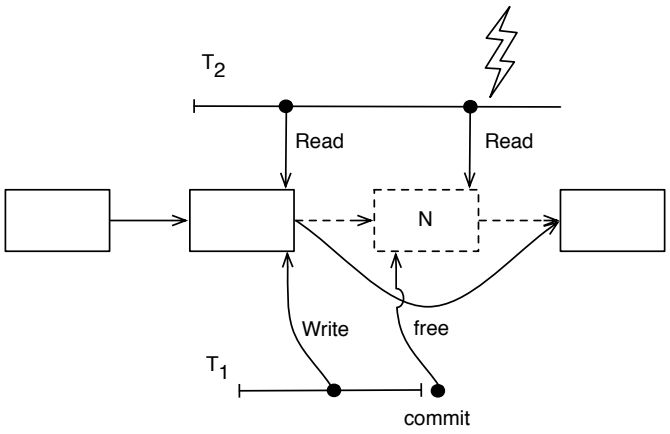


Figure 4.9: Example of a problem that can occur if deallocations are performed at the commit-time.

Transactional memory allocation is simpler to support of the two. When a transaction allocates a memory block, the block is allocated using the underlying non-transactional memory allocator and the address of the memory block is logged. If the transaction commits, its allocation log is simply discarded. If it aborts, the performed allocations need to be rolled back, which is done by traversing the allocation log and deallocating all memory blocks in it.

Similarly, when a transaction deallocates a memory block, the deallocation is logged. Unlike allocations, the deallocations are not performed immediately: the deallocated memory might get reallocated and its contents changed after the deallocation, which makes the rollback of the deallocations difficult. Instead, the deallocations are postponed until the transaction commits. If the transaction aborts, its deallocation log is simply discarded and none of the blocks gets actually deallocated. If the transaction commits, then the deallocations need to be performed. However, it is not safe to perform the deallocations immediately on commit, as some other transactions might still be referencing the deallocated block. This is a well known problem with memory deallocation when readers are invisible [29, 46, 60, 80], and is briefly mentioned in the previous chapter. Basically, the deallocating transaction cannot be sure, at commit-time, that no other transaction is about to access the memory block after it commits.

Figure 4.9 illustrates that a reader transaction can indeed access deallocated memory if the deallocations are performed immediately on commit. In the example, two transaction are accessing a linked list. Transaction T_1 is unlinking node N from the list, whereas transaction T_2 is traversing the list. In the example, T_2 obtains a pointer to N and is then blocked for a while before dereferencing it. Then, T_1 unlinks N from the list, successfully commits, and deallocates N immediately after commit. If the next step that T_2 performs, after resuming, is the memory access at line 23 in Figure 4.3, it will read deallocated memory previously occupied by N. This can cause the program to crash, if the memory has already been returned to the operating system and unmapped from the process's address space. Other problems related to

memory management can occur if some of the data accesses to *N* are not transactional, which is often done for performance. Examples of such usage are discussed in the previous chapter.

To ensure that the deallocated memory will not be accessed by a concurrent transaction, the deallocation needs to be postponed until all transactions referencing the memory block complete, either by committing or aborting. SwissTM takes a coarser-grained approach, similar to the ones in [46, 66], and postpones the deallocation until all transactions concurrent to the deallocating transaction have completed, regardless of whether they are referencing the memory to be deallocated or not. Furthermore, the deallocating transaction does not need to wait until all concurrent transactions actually complete. Instead, it is enough to wait until the other transactions have observed the changes performed by the deallocating transaction. Note that the deallocating transaction unlinks the deallocated objects from all shared data structures thus preventing concurrent transactions that have observed its changes from accessing the deallocated memory. Transactions' validity timestamps are used to determine when it is safe to perform the actual deallocations: when all validity timestamps become higher or equal to the commit timestamp of the deallocating transaction, that means that all other transactions have observed the deallocating transaction's updates, making it safe to deallocate the memory. To avoid postponing deallocations indefinitely if some thread stops executing transactions, the validity timestamps are set to a special value on commits and aborts, indicating that their thread is not executing any transaction at the moment.

To optimize the memory deallocations, a thread does not wait to perform the actual deallocations after the deallocating transaction commits. Instead, it postpones the deallocations, by putting them in a thread-local log together with the transaction's commit timestamp, and continues the execution. Periodically, the thread traverses this log and performs all deallocations that have become safe, in the following way. The log is organized as a linked-list of arrays, similarly to the log data structure described above. Each of the arrays stores a number of pointers to memory blocks for deallocation and the commit timestamp of the last transaction that stored pointers in the array. When a transaction commits, it stores the pointers to the deallocated memory blocks into the current deallocation array. If the array becomes full, the transaction writes its commit timestamp to the array, moves it to the list of full arrays, and obtains a new array for the future transactions. It also attempts to deallocate the memory blocks stored in the list of full arrays at this time. To deallocate the memory, the thread reads validity timestamps from transaction descriptors of all threads in the system and finds the minimum value. It traverses the list of full arrays and frees all memory blocks pointed to by the arrays that have the timestamps lower than or equal to the minimum observed validity timestamp. In this way, the deallocations are buffered to reduce the overheads of frequently determining the minimum validity timestamp and traversing the list of full arrays. The downside of buffering is that the memory requirements of the application are temporarily increased, but this does not cause any problems in practice, even with large-scale STMBench7 workloads.

Supported platforms. I built SwissTM to be portable, using standard C/C++ integer and pointer types, standard language and system library calls, and `atomic_ops` [10] library for portable support of atomic instructions on different systems. This resulted in code that is largely platform-independent. I built SwissTM versions for Linux and MacOS on x86 and Solaris on SPARC, with support for both 32-bit and 64-bit versions for x86. SwissTM can be ported to other platforms as needed, with just a little additional effort.

4.3 Evaluation

In this section, I present an extensive evaluation of SwissTM, which compares performance of SwissTM to performance of RSTM [77], TL2 [28], and TinySTM [43] on a range of benchmarks: STMBench7 [52], STAMP [17], Lee-TM [8], and the red-black tree micro-benchmark. I first describe the benchmarks in detail, and then present and discuss the results.

4.3.1 Benchmarks

The benchmarks I used for the experiments represent a large spectrum of workload types: from simple data structures with small transactions, which characterize the red-black tree micro-benchmark, to complex applications with possibly long transactions, which characterize STMBench7.

STMBench7. Section 3.3 describes STMBench7 in detail. Here, I summarize its most important features to contrast it to the other benchmarks used in the evaluation. STMBench7 [52] is a synthetic benchmark that aims at representing realistic, complex, object-oriented applications. It exhibits a large variety of operations, ranging from very short, read-only operations to very long ones that modify large parts of the data structure, and workloads, ranging from workloads consisting mostly of read-only transactions to workloads dominated by update transactions. The data structure used by STMBench7 is many orders of magnitude larger than in other typical STM benchmarks. Also, its transactions are longer and access more objects.

STMBench7 is inherently object-based and, also, its implementations use standard language libraries. A thin wrapper that implements object-based interface on top of the word-based one, described in Section 3.3, is thus necessary to use STMBench7 with word-based STMs, such as TL2, TinySTM, and SwissTM.

STAMP. STAMP [17] is a TM benchmarking suite that consists of eight transactional programs, which can be configured to represent different workload characteristics. In the experiments, I used STAMP 0.9.9 and the 10 workloads defined in its distribution. STAMP applications are representative of various real-world workloads, including bioinformatics, engineering, computer graphics, and machine learning workloads. While STAMP covers a

broad range of possible STM uses, its does not involve transactions as long and as complex as transactions of STMBench7. Also, some STAMP benchmarks use fine-grained transactional synchronization, which is not typical for code that might be produced by average, non-expert programmers or generated automatically by a compiler, for example along the lines of automatic mutual exclusion [1]. Furthermore, some STAMP algorithms, such as *bayes*, split logical operations into multiple transactions and use intricate programming techniques that are not representative of average programmers' skills. Because of the different characteristics, STAMP and STMBench7 complement each other well when evaluating STM performance.

Next, I briefly describe the benchmarks and workloads that constitute STAMP. More details are available in [17].

Bayes implements an algorithm for learning the structure of a bayesian network, using hill-climbing search with a combination of local and global search. It uses long transactions, with large read- and write-sets.[3] Most of the execution time is spent inside transactions, which encounter high contention. Transactions in *bayes* also combine transactional and non-transactional accesses to shared data, exploiting the relaxed nature of the algorithm, which is not representative of how average programmers write code.

Genome performs genome sequence matching. It does so in two phases: the threads first eliminate duplicate DNA segments from the input, and then they match the remaining segments. All string manipulations are performed using non-transactional accesses to improve performance. *Genome* uses transactions of medium length, with medium-sized access sets. Threads spend most of the execution time inside transactions, but the transactions encounter relatively low contention.

Intruder emulates signature-based network intrusion detection. To detect intrusion attempts, it compares network packets to signatures of past network intrusions. The threads obtain the network packets from a single shared queue, which turns out to be a source of high contention, especially at higher concurrency levels. The transactions are short in duration, they access a medium number of memory locations, but they encounter high contention. Threads spend a medium amount of time in transactions.

Kmeans implements the K-means clustering algorithm, that partitions input data into K similar sets. Input data is processed in several iterations with threads synchronizing on barriers between iterations. *Kmeans* uses very short transactions that access only a few locations: the most complex transaction accesses only four locations. Most of the processing time is spent outside of the transactions. I used two different *kmeans* workloads: a high-contention and a low-contention one, denoted as *kmeans-high* and *kmeans-low*, respectively.

Labyrinth is a circuit-routing program that is used to find the shortest paths between points in a three-dimensional maze. It uses Lee's algorithm [98] and is quite similar to Lee-TM [8]

[3] Whereas the *bayes* transactions are long and have large access sets with respect to other STAMP benchmarks they are still much shorter and access much fewer objects than STMBench7 transactions.

Workload	Parameters
bayes	`-v32 -r4096 -n10 -p40 -i2 -e8 -s1`
genome	`-g16384 s64 -n16777216`
intruder	`-a10 -l128 -n262144 -s1`
kmeans-high	`-i random-n65536-d32-c16.txt -m15 -n15 -t0.00001`
kmeans low	`-i random-n65536-d32-c16.txt -m40 -n40 -t0.00001`
labyrinth	`-i random-x512-y512-z7-n512.txt`
ssca2	`-s20 -i1.0 -u1.0 -l3 -p3`
vacation-high	`-n4 q60 -u90 -r1048576 -t4194304`
vacation-low	`-n2 -q90 -u98 -r1048576 -t4194304`
yada	`-a15 -i ttimeu1000000.2`

Table 4.2: STAMP workloads.

benchmark described below. The code and the inputs are different from Lee-TM, and that is why I include both benchmarks in the evaluation. The transactions *labyrinth* uses are long, with large access sets, but, in my experience, they do not experience particularly high contention. Most of the processing is done inside transactions.

Ssca2 benchmark implements the first kernel from a set of four computational kernels with the same name, that operate on multi-graphs. The kernel builds an efficient graph using adjacency arrays, with transactions being used only to synchronize the access to the arrays. Such access patterns result in short transactions with small read- and write-sets that experience little contention. Consequently, most of the processing is performed outside of the transactions.

Vacation emulates an on-line transactional system that supports a travel agency, with the data stored in several tree data structures. The program is similar to SPECjbb2000 [110] benchmark. *Vacation* uses transactions of medium length that perform a medium number of memory accesses, but most of the execution time is spent inside transactions. Similarly to *kmeans*, there are two *vacation* workloads: a high-contention *vacation-high* and a low-contention *vacation-low* workload. In my experience, though, threads encounter little contention in both of them.

Yada is a program for Delaunay mesh refinement that operates on a graph of mesh triangles, which gets continuously updated by different threads during program execution. Transactions in *yada* are large, with large read- and write-sets and they experience medium contention. Most of the execution time is spent inside transactions.

Table 4.2 lists all the STAMP workloads used in the experiments and specifies the input parameters for each of them.

Lee-TM. Lee-TM [8] is a benchmark based on Lee's circuit routing algorithm [98] that exposes large, realistic workloads. The algorithm takes pairs of locations on an integrated circuit as its input and produces non-intersecting routes between them. While transactions of Lee-TM

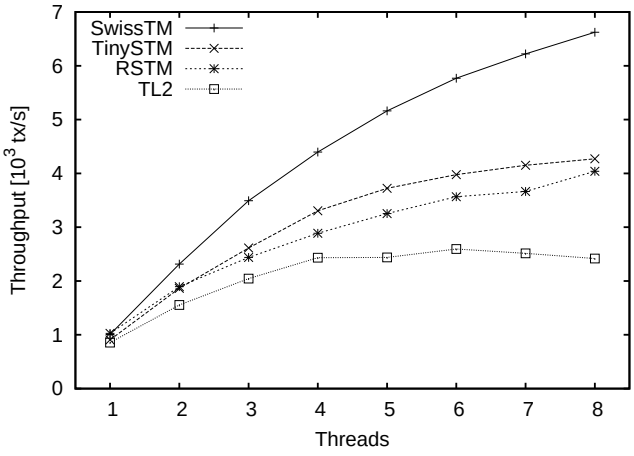
are rather large, they exhibit very regular access patterns: every transaction first reads a large number of locations while searching for a suitable path, and then updates a small number of them while setting the path. Moreover, the benchmark uses a simple data structure, which is, in essence, a matrix of integer variables representing the state of the points on the integrated circuit. As mentioned above, STAMP contains a benchmark, called *labyrinth*, that uses the same algorithm as Lee-TM. However, Lee-TM comes with real-world input sets that make it more realistic than *labyrinth*. Lee-TM software distribution includes two large input data sets: *memory* and *main* circuit boards.

Red-black tree. Most work on STMs is evaluated using various micro-benchmarks. The red-black tree micro-benchmark was first used in the initial paper on dynamic STMs [63], to demonstrate the flexibility of STM, and has been used for evaluation of various STMs ever since. In the benchmark, a single shared red-black tree is used to implement an integer set. Each transaction executes a single lookup, insert, or remove of a randomly chosen integer from the set. The integers are chosen from a predefined range of values. Initially, half of the integers from the range are inserted into the tree. The ratio of insert and remove operations is the same, which keeps the tree size roughly constant during the execution, with the size equal to half of the selected range of values. The level of contention between threads can be modified by changing the ratios of executed operations and the range of values from which the integers are selected. In the experiments, I used the range of 16,384 values and a mix of 80% lookup and 20% update operations. The short and simple transactions of the micro-benchmark represent a good test of STM mechanics and are good for comparing low-level details of various implementations. The red-black tree micro-benchmark is included in the evaluation to measure the impact of supporting SwissTM's quite sophisticated conflict detection and contention management policies on its performance with short transactions.

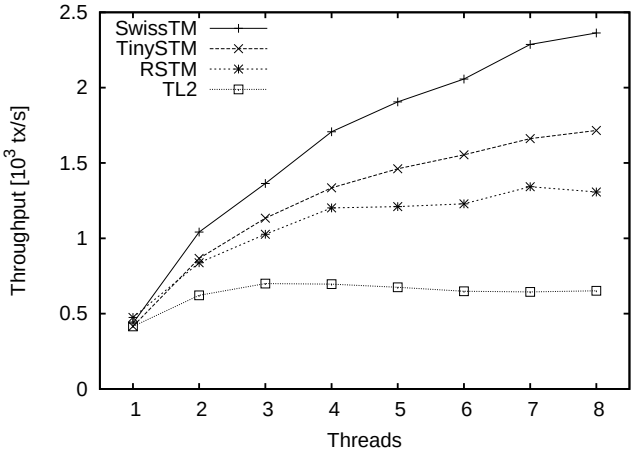
4.3.2 Experimental settings

All measurements were performed on the system used in the previous chapter: a system with four dual-core AMD Opteron 8216 CPUs at 2.4 GHz, with 1024 KB cache, for a total of eight cores. The system has 8 GB of RAM and runs Linux operating system. The results are averaged over multiple runs, where the length and the number of runs were chosen to reduce variations in collected data. I used 20 runs for STMBench7 and STAMP, 10 runs for LeeTM, and 80 runs for the red-black tree micro-benchmark.

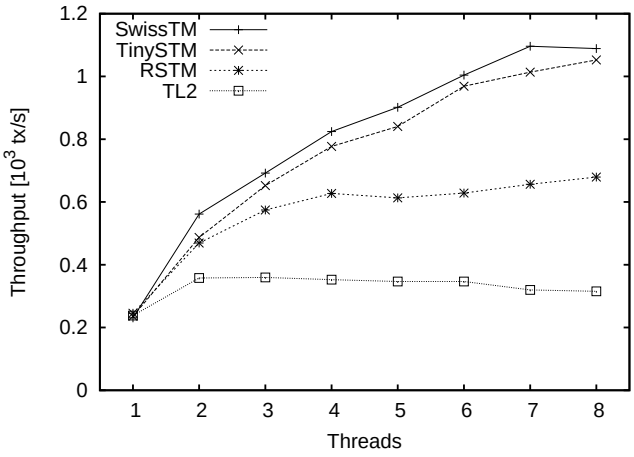
I used the TL2 x86 implementation provided with the STAMP benchmark suite version 0.9.9. I used RSTM version 3 and TinySTM version 0.9.5 available from the respective web pages. The STMs were configured to obtain best performance on the used benchmarks. Unless stated otherwise, RSTM was configured to use eager conflict detection, invisible reads with the commit counter heuristic, and the Polka contention manager. I used the default configuration of TL2 with deferred updates, lazy conflict detection, Timid contention manager, and GV4 al-



(a) Read dominated



(b) Read-write



(c) Write dominated

Figure 4.10: Throughput of STMBench7 with SwissTM, RSTM, TL2, and TinySTM.

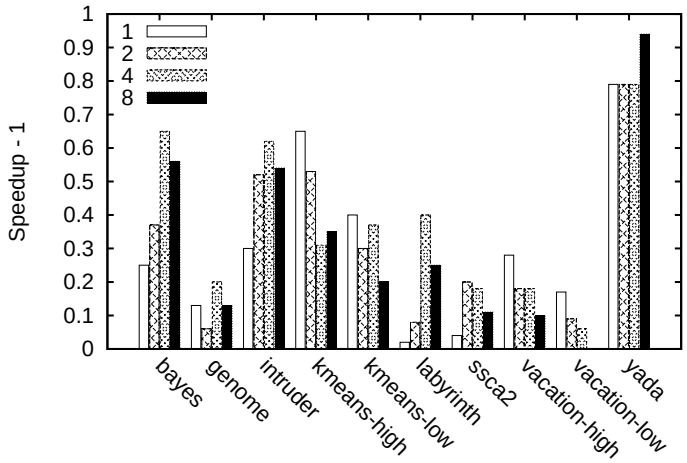
gorithm [69] for incrementing the shared time-base. Similarly, I used the default configuration of TinySTM with in-place updates, encounter-time locking, and Timid contention manager.

Similarly to the previous chapter, I could not use the original TL2 implementation as it does not support x86 architecture, which I used in the experiments. Also, the original TL2 does not support transactional memory management in a straightforward manner, and is, therefore, difficult to use with benchmarks I used in the evaluation without significant changes to the benchmark code. While I did not use the original TL2 implementation in a setting that it was primarily designed for, I believe that the TL2 for x86 is the best representative of the TL2 algorithm and design available for x86 architecture.

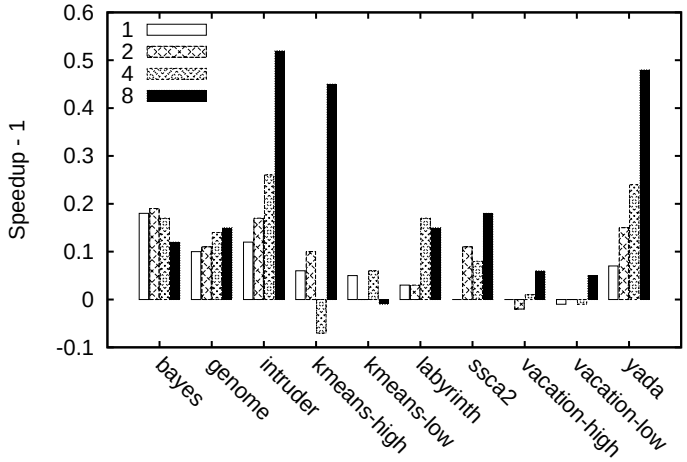
4.3.3 STMBench7

Figure 4.10 shows the performance of all used STMs on STMBench7. In this experiment, RSTM was configured to use the Serializer contention manager, as this resulted in the best performance. The figure shows that SwissTM outperforms all other STMs on both read-dominated and read-write workloads by a significant margin, while also achieving superior scalability. SwissTM achieves 55% higher throughput than TinySTM, which performs best of the other STMs, on the read-dominated workload and almost 40% higher throughput on the read-write workload. The difference in achieved performance with respect to other STMs is higher: SwissTM outperforms RSTM by 65% and TL2 by 175% on the read-dominated workload, and RSTM by 80% and TL2 by more than 250% on the read-write workload. Furthermore, SwissTM also achieves the highest throughput in the high-contention write-dominated workload, but it is only marginally faster than TinySTM, achieving about 5% higher throughput. SwissTM outperforms the other two STMs by considerable margins even on the write-dominated workload: it achieves about 60% higher throughput than RSTM and about 200% higher throughput than TL2.

The main reason for the good performance of SwissTM is (1) its optimism in detecting read-write conflicts when compared to RSTM and TinySTM, and (2) its conservatism in detecting write-write conflicts when compared to TL2. The contention management scheme used by SwissTM also helps boost performance, as illustrated in Section 4.4. The reason for TinySTM having almost the same performance as SwissTM on the write-dominated workload is that SwissTM's more optimistic policies are better suited to low-contention than high-contention workloads, such as the write-dominated STMBench7. TL2 performs poorly even in the read-dominated workload: it does not scale after four threads and its performance becomes even worse on higher-contention workloads. The main reason for this is its use of the lazy conflict detection scheme, which wastes more work performed by aborted transactions than the eager write-write conflict detection used by other STMs.



(a) TL2



(b) TinySTM

Figure 4.11: SwissTM compared to TL2 and TinySTM on STAMP.

4.3.4 STAMP

Figure 4.11 compares the performance of SwissTM, TL2, and TinySTM on the STAMP workloads. Due to API incompatibility, I could not implement STAMP with the version of RSTM I used: RSTM is object-based, whereas STAMP requires a word-based interface, which makes it difficult to simply plug-in RSTM into STAMP. Instead, STAMP programs would have to be fully reimplemented to use the object-based interface, which is outside of the scope of this thesis. The figure depicts the speedup of SwissTM compared to TL2 and TinySTM, calculated as $Speedup = Duration_{OtherTM} / Duration_{SwissTM}$ and presented as $Speedup - 1$, which means that SwissTM outperforms TL2 and TinySTM for positive values in the figure and performs worse for negative values.

As the figure shows, SwissTM outperforms TL2 on all STAMP workloads, for all thread counts. SwissTM outperforms TL2 by over 50% with eight threads on the *bayes*, *intruder*, and *yada* workloads, achieving almost twice as high performance as TL2 on *yada*. The difference in achieved performance is smaller on the other benchmarks, but is still significant: SwissTM outperforms TL2 by over 20% on *kmeans-high*, *kmeans-low*, and *labyrinth*, and by about 10% on *genome*, *ssca2*, and *vacation-high*. SwissTM performs only slightly better than TL2 on *vacation-low*, having about 5% better performance.

Similarly, SwissTM achieves better performance than TinySTM on STAMP, although the observed difference in performance is smaller than with TL2. With eight threads, SwissTM outperforms TinySTM on all STAMP workloads, with the exception of *kmeans-low* where the performance is roughly the same, as TinySTM achieves only about 1% higher performance. SwissTM outperforms TinySTM by over 45% with eight threads on *intruder*, *kmeans-high*, and *yada*, and by over 12% on *bayes*, *genome*, *labyrinth*, and *ssca2*. It is only slightly faster on the two *vacation* workloads where it achieves about 5% better performance.

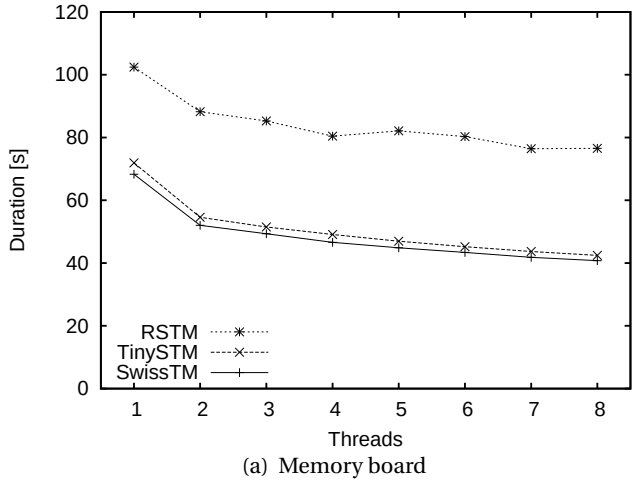
To summarize, SwissTM outperforms both TL2 and TinySTM in all STAMP workloads except one, TL2 by up to 100% and TinySTM by up to 45% with eight threads. In the only workload where it does not outperform TinySTM, the difference in performance is only 1%, which is negligible. It also achieves good performance with lower thread counts, and it scales well as the number of concurrent threads increases.

4.3.5 Lee-TM

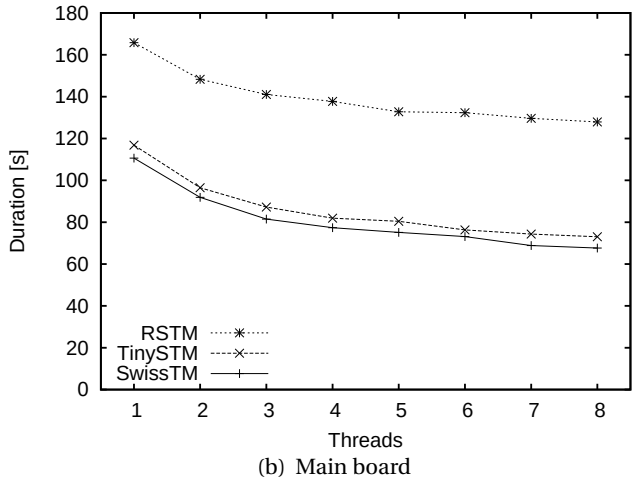
Figure 4.12 compares the performance of SwissTM, RSTM, and TinySTM on the Lee-TM benchmark, depicting the time Lee-TM takes to complete the execution with different STMs. I do not show the results of Lee-TM with TL2 as the executions with the version of TL2 I had at my disposal do not complete even with a single thread, due to a bug in the implementation.

The results show that RSTM has by far the lowest performance. The main reason for its poor performance are high overheads of accessing simple objects used in Lee-TM. The objects consist of a single integer variable, and treating integer variables as full-fledged objects introduces significant overheads compared to accessing them directly with the word-based STMs. For this reason, RSTM implementation of Lee-TM takes almost twice as long to complete as SwissTM and TinySTM implementations. SwissTM and TinySTM have very similar performance, with SwissTM being faster by a small margin of between 4% and 8% for all thread counts.

These results show that the more sophisticated conflict detection and contention management policies of SwissTM do not matter as much with Lee-TM as with STMBench7, because transactions access data in uniform patterns and there is little contention. Interestingly, Lee-TM results demonstrate that the implementation cost of relatively sophisticated policies in SwissTM is not necessarily high, as SwissTM performs better than TinySTM on Lee-TM even with a single thread.



(a) Memory board



(b) Main board

Figure 4.12: Execution time of Lee-TM benchmark with SwissTM, RSTM, and TinySTM.

4.3.6 Red-black tree

Figure 4.13 compares the performance of SwissTM, TL2, TinySTM, and RSTM on the commonly used red-black tree microbenchmark. Similarly to Lee-TM results, RSTM delivers significantly lower performance than the other three STMs, due to high overheads of accessing simple objects. Such low-level overheads have most significant impact on micro-benchmarks like this one, resulting in RSTM achieving 4x lower throughput than SwissTM.

The higher overheads of accessing single memory locations are the reason for SwissTM having lower performance than TinySTM and TL2 with fewer than three threads: TL2 and TinySTM use only one lock for each memory location, whereas SwissTM uses two, which results in slightly higher overheads. It is worth noting that red-black tree is the only benchmark for which slightly higher overheads of SwissTM having two locks in *orecs* have more than negligible performance impact.

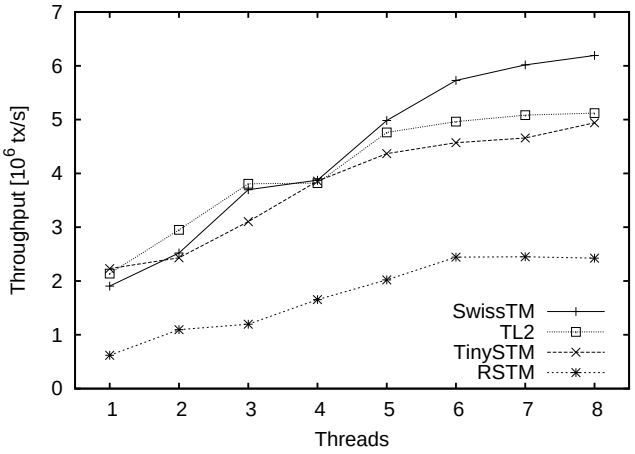


Figure 4.13: Throughput of SwissTM, TL2, TinySTM, and RSTM on red-black tree.

SwissTM exhibits better scalability than TinySTM and TL2 and, despite having 17% and 12% lower single-threaded performance respectively, it outperforms them with more than four threads. Furthermore, it achieves 25% higher throughput than TinySTM and 20% than TL2 with eight threads. The experiment with the red-black tree micro-benchmark shows that even with relatively sophisticated policies SwissTM achieves good performance and scalability in small-scale workloads that accentuate the low-level overheads, if implementation uses efficient techniques and data structures.

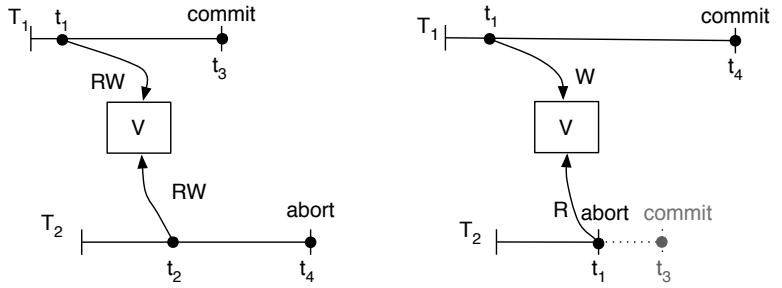
4.4 Dissecting SwissTM

The extensive evaluation of SwissTM's performance in the previous section shows that SwissTM indeed performs well across a wide-range of workloads. In this section, I evaluate the design choices underlying SwissTM, by comparing them to their most prominent alternatives. I individually evaluate: the mixed eager-lazy conflict detection strategy, the Two-phase contention manager, and the locking granularity.

4.4.1 Conflict detection

Current state-of-the-art STMs typically detect both read-write and write-write conflicts in the same way, either eagerly as soon as conflicts occur, such as TinySTM, McRT-STM, and Bartok STM, or lazily at commit time, such as, for instance, TL2. Detecting conflicts eagerly helps avoid wasting work of transactions that are doomed to abort after a conflict. Lazy conflict detection, on the other hand, is more optimistic and gives transactions more possibilities to commit. Figure 4.14 illustrates the advantages and limitations of both approaches.

Figure 4.14(a) depicts an execution of an STM that uses the lazy conflict detection. In the figure, transaction T_2 spends time between t_3, which is the commit time of T_1, and t_4, which is



(a) Lazy (b) Eager

Figure 4.14: Limitations of pure lazy and eager conflict detection strategies.

the commit time of T_2, performing work that is doomed to be rolled back. The period between t_3 and t_4 can be significant if transactions are long. It is worth noting that both T_1 and T_2 could commit in such an execution with an STM that uses the lazy conflict detection, if they both only write to V. However, pure write-write conflicts are typically rare, as transactions usually first read some data and then update it later. Because of this, STMs that use lazy conflict detection react too slowly to write-write conflicts, which are good signs that the conflicting transactions cannot proceed in parallel, and this results in transactions performing work that has to be rolled back later. Figure 4.14(b), shows an example execution of an STM that uses eager conflict detection. There, transaction T_2 is aborted at time t_1 and has to wait until time t_4 before continuing, although it could commit already at time t_3 if lazy scheme was used. The waiting time of T_2 might be significant if T_1 is very long.

As discussed, SwissTM takes the best of both strategies: it detects write-write conflicts eagerly and read-write conflicts lazily. This combined strategy is beneficial for complex workloads with long transactions because it (1) prevents transactions with write-write conflicts from running for a long time before detecting the conflict, and (2) allows short transactions having a read-write conflict with longer ones to proceed, thus increasing parallelism.

The comparison of several conflict detection variants presented in Figure 3.4 in the previous chapter, clearly shows that eager conflict detection is better than lazy for large-scale workloads, which is confirmed by the experiments presented in the previous section. Next, I empirically outline the scenarios in which the mixed eager-lazy conflict detection is better than the pure eager conflict detection.

To compare the eager-lazy and pure eager schemes, I modified the Lee-TM benchmark and compared the performance of SwissTM and TinySTM on the modified benchmark. The performance of the original Lee-TM does not seem to be significantly impacted by the choice of the conflict detection and contention management schemes, because the transactions in Lee-TM are very regular: they first read a large number of locations and then update a few of those locations. To change this, I introduced a dose of irregularity in Lee-TM by adding a single object O_c that every transaction reads at its start. In addition, each transaction updates

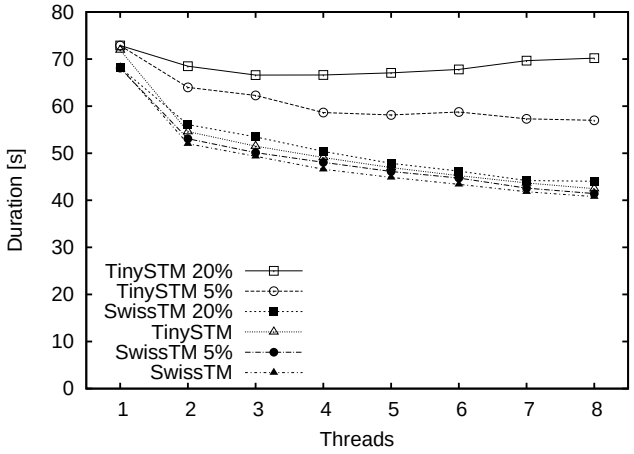


Figure 4.15: Execution time of SwissTM and TinySTM in "irregular" Lee-TM benchmark with *memory* circuit board input data set.

O_c at its start with a small probability R, thus probabilistically causing a read-write conflict with all other transactions.

Figure 4.15 compares the performance of TinySTM and SwissTM when R is 0%, 5%, and 20%. It reveals that TinySTM is very sensitive to the long-lasting read-write conflicts among transactions, as its performance degrades significantly even when R is only 5%. Worse, when R is 20% TinySTM does not scale to more than three threads. In contrast, SwissTM's performance degrades only slightly even when R is 20%, with performance still scaling well as the number of threads increases. The contention manager used by SwissTM does not impact the performance on the "irregular" Lee-TM much, as only a few write-write conflicts are caused for small values of R. Therefore, the main reason for SwissTM's resiliency to the long-lasting read-write conflicts in the "irregular" Lee-TM is the manner in which they are detected. With lazy detection of read-write conflicts, if the reader transactions attempts to commit before the writer, it succeeds, thus increasing the parallelism among transactions. In contrast, with eager detection of read-write conflicts, a single transaction that writes to O_c prevents all other transactions from committing before it commits or aborts.

These experiments illustrate that applications exhibiting regular access patterns benefit the most from lowering single-location access costs and are not impacted significantly by the policy implemented by the conflict detection scheme. However, for applications where the access patterns introduce even small irregularities, especially those creating long-lasting read-write conflicts, SwissTM's optimistic approach yields significant benefits. Combining eager write-write and lazy read-write conflict detection enables SwissTM to take the best of the both worlds and achieve good performance across a range of workloads.

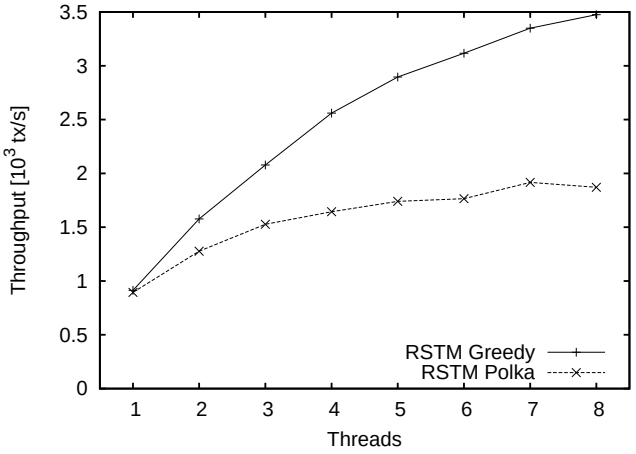


Figure 4.16: Best STMBench7 read-dominated throughputs achieved by RSTM with Polka and Greedy contention managers.

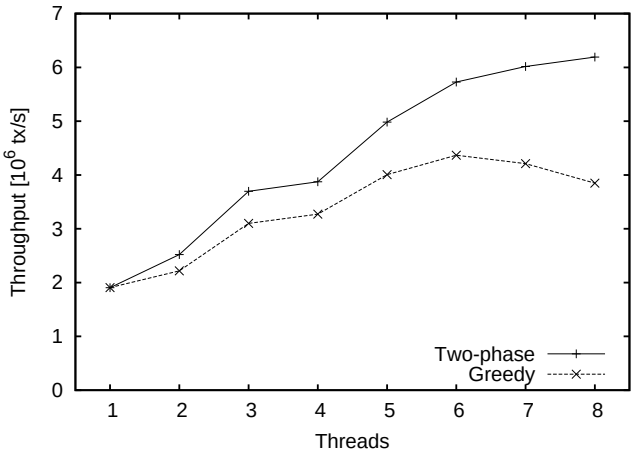


Figure 4.17: Throughput of SwissTM with the Two-phase contention manager and with Greedy on the red-black tree.

4.4.2 Contention management

Figure 4.16 shows that, on STMBench7, RSTM using Greedy performs better than when using Polka contention manager, similarly to what is shown in the previous chapter. Similarly, Greedy outperforms other contention managers provided in RSTM distribution [100]. For this reason, SwissTM uses Greedy for large transactions, such as the ones in STMBench7.

However, Greedy performs poorly in workloads where there are many short transactions. The reason for this is the implementation of Greedy: each transaction atomically increments the single shared counter at its start, to obtain the contention management timestamp. Greedy uses these timestamps to totally order the transactions, which allows it to abort the younger transaction in case of a conflict. With short transactions, however, threads keep constantly updating the counter, significantly increasing the contention and the number of cache misses.

This impacts the performance adversely, and the counter becomes the scalability bottleneck. Figure 4.17 compares the performance of SwissTM with the Two-phase contention manager and with Greedy on the red-black tree micro-benchmark. The performance of the two SwissTM variants is virtually identical with a single thread. However, the Greedy variant is slower already with two threads and it scales poorly: with more than six threads, the performance starts to degrade. This problem is hard to notice with longer transactions as the overheads caused by the increase in the rate of cache misses is relatively small compared to the other work performed by transactions. As shown in Figure 4.17, the Two-phase contention manager completely solves the problem, improving both performance and scalability over Greedy, and achieving 60% higher throughput than Greedy with eight threads. This is because it allows all short and read-only transactions to commit without incrementing the shared counter used by the Greedy algorithm, thus greatly reducing contention on the shared counter. Yet, the Two-phase contention manager handles conflicts among long transactions as efficiently as Greedy.

A natural question is one of whether the second phase of the Two-phase contention manager is necessary at all, or is the Timid contention manager appropriate for all transactions. After all, Figures 4.13 and 4.17 demonstrate that SwissTM performs very well even when there are only a few transactions that transition to the second phase of contention management. To answer this question, I compare performance of SwissTM with the Two-phase and Timid contention manager on STMBench7. Figure 4.18 shows the results of the comparison for all three STMBench7 workloads with 1 to 8 threads. It depicts speedup of SwissTM when using the Two-phase contention manager compared to Timid, calculated as $Speedup = Throughput_{Two\text{-}phase} \Big/ Throughput_{Timid}$. The comparison shows that SwissTM with the Two-phase contention manager achieves up to 16% higher throughput than with Timid in the high-contention, write-dominated workload. This difference is, expectedly, lower in the lower-contention workloads where there are fewer conflicts: on read-dominated workload SwissTM with Two-phase contention manager outperforms SwissTM with Timid by at most 3% and on read-write workload by at most 9%.

These experiments show that both phases of the Two-phase contention manager indeed help SwissTM efficiently support mixed workloads, as the different phases improve its performance on different types of workloads.

SwissTM uses linear back-off on aborts, proportional to the number of successive transaction restarts. It might seem beneficial, however, to have transactions restart immediately after rollback, as the waiting just decreases the reaction time before the transaction is restarted. TinySTM, for example, adopts the approach of immediate restarts. However, restarting immediately tends to increase contention on cache lines containing data that get updated very frequently. Consequently, short back-offs after transaction rollbacks can improve performance, as discussed in Section 3.4. To evaluate the effects of the back-offs, I compared the performance of SwissTM with and without back-off on the *intruder* benchmark from STAMP. I chose the *intruder* benchmark, as it is a good candidate to show the impact of the back-offs:

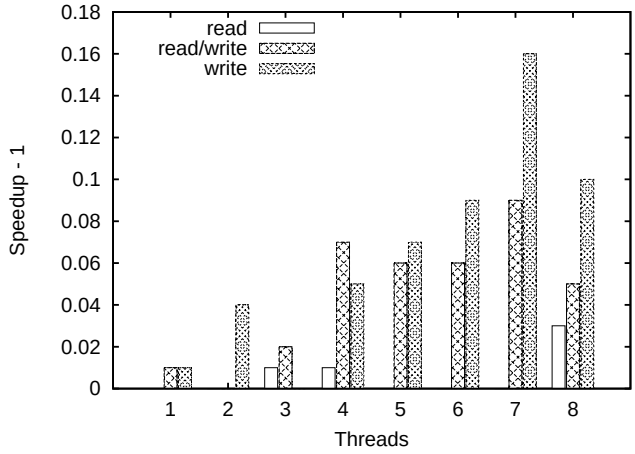


Figure 4.18: Comparison of SwissTM performance with the Two-phase contention manager and with Timid on STMBench7.

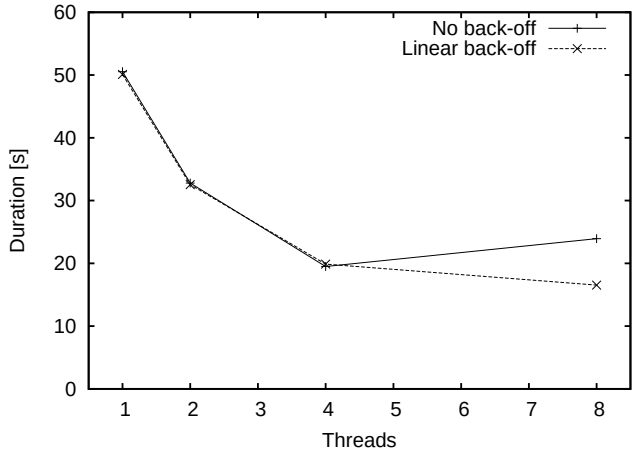


Figure 4.19: Execution time of *intruder* with SwissTM with and without back-off on transaction restart.

it contains a "hot spot", as transactions repeatedly dequeue elements from the same shared queue. Figure 4.19 shows the results of the experiment. The results convey that the immediate restart can indeed cause scalability problems with higher thread counts, as the performance with eight threads is lower than with four, when transactions restart immediately after aborting. As the results also show, the simple randomized linear back-off scheme resolves this scalability issue and has about 45% better performance with eight threads than the scheme with no back-off.

4.4.3 Locking granularity

An important implementation choice when building a new STM is the *orec* table configuration, in particular the size of the memory stripe that gets mapped to each *orec*. Increasing the size

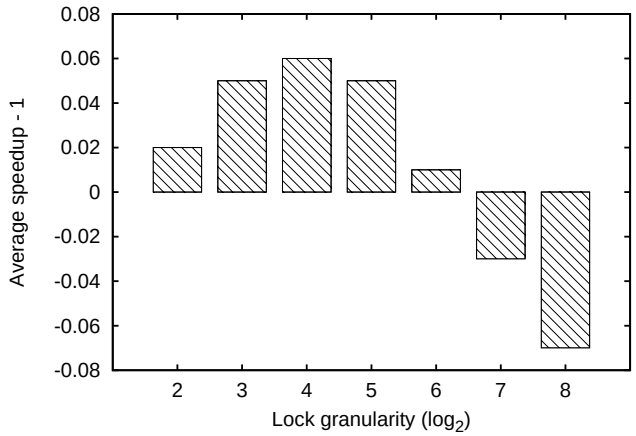


Figure 4.20: Average speedup across all benchmarks used, with one subtracted, of locking granularities from 2^2 to 2^8 compared to all other granularities, when using eight threads.

of the memory stripe reduces locking and validation time, as it improves data access locality, but it also increases the rate of aborts, as the number of false conflicts increases with the size of the stripe. The optimal value for the stripe size is application specific, and different STMs adopt different stripe sizes. For example, TinySTM and TL2 use the stripe size of one word, whereas McRT-STM uses the the stripe size of the whole cache line.

While implementing SwissTM, I used the available benchmarks to experimentally determine the best stripe size for SwissTM. Figure 4.20 compares the performance of several locking granularities. Each granularity is a power of two, as this significantly simplifies address-to-lock mapping, as described in Section 4.2.4. For each granularity, the figure depicts the average speedup, with one subtracted, of that particular granularity over all other granularities used with eight threads. The figure shows performance with the highest number of threads available in order to increase the probability of false conflicts and avoid underestimating their effects. The experiments were performed on a 32-bit architecture, which means that the word size is 2^2 bytes. The results show that SwissTM performs best with granularity of 2^4 bytes, while achieving only slightly lower performance with granularities of 2^3 and 2^5 bytes. It is interesting to note that SwissTM with the commonly used stripe sizes of one word and one cache line, which is 2^6 bytes, has, respectively, 4% and 5% lower performance on average than with the selected stripe size of 2^4 bytes. The results also convey that using coarser locking granularities adversely impacts performance: SwissTM with stripe size of 2^8 bytes has 20% lower performance than with the stripe size of 2^4 bytes. The breakdown of the performance impact of locking granularity across benchmarks is given in Table 4.3.

An interesting conclusion based on these results is that, whereas using different locking granularities does impact performance, the impact of using sub-optimal stripe size is usually not significant, being in the order of several percent. Also, even when using coarser locking

	Difference in performance					
	2^4 vs. 2^2	2^4 vs. 2^3	2^4 vs. 2^5	2^4 vs. 2^6	2^4 vs. 2^8	2^2 vs. 2^6
bayes	0.16	0.45	0.42	0.81	0.92	0.57
genome	0.13	0.04	-0.01	−0.03	0.12	−0.14
intruder	0	-0.02	-0.04	−0.04	0.04	−0.04
kmeans-high	0.19	0.1	0.04	0.4	1.67	0.18
kmeans-low	0.14	-0.11	-0.03	0.05	0.45	−0.08
labyrinth	−0.12	-0.14	-0.07	−0.09	-0.14	0.04
ssca2	0	0	0	0	-0.01	0
vacation-high	0.14	0.05	-0.02	−0.03	0.01	−0.15
vacation-low	0.12	0.04	-0.03	−0.05	-0.08	−0.15
yada	0	0.12	0.02	-0.03	0.21	0.12
rbtree	−0.01	0	-0.03	0	0.15	0.01
leetm-memory	0.01	-0.01	-0.01	−0.03	-0.02	−0.04
leetm-main	0.02	-0.04	0	−0.01	0.01	−0.02
sb7-read	0	0	-0.04	−0.02	-0.04	−0.02
sb7-read-write	0	−0.01	-0.01	−0.03	-0.03	−0.02
sb7-write	−0.04	-0.02	-0.05	−0.06	-0.03	−0.02
Average	0.05	0.02	0.01	0.06	0.20	0.01

Table 4.3: Comparing several different locking granularities. The values represent relative speedups, with one subtracted, when using eight threads.

granularities the rate of false conflicts is not increased to the level where it prevents SwissTM from scaling.

To avoid any confusion, it is worth pointing out that in all experiments presented so far, SwissTM uses the same locking granularity of 2^4 bytes.

4.5 Extending SwissTM

In this section I describe two extensions to base SwissTM: integration of SwissTM with standard compilers and support for the privatization idiom.

4.5.1 Compiler support

To truly support a simple programming model SwissTM needs to be integrated with an STM compiler. Otherwise, the use of the low-level interface exposed by SwissTM can be tedious and can lead to subtle and hard-to-detect bugs if programmers, mistakenly, do not perform all accesses to shared data using STM. To make SwissTM easier to use, I integrated it with Intel's C/C++ compiler [85], by implementing the compiler's ABI [67] on top of the SwissTM's word-based interface.

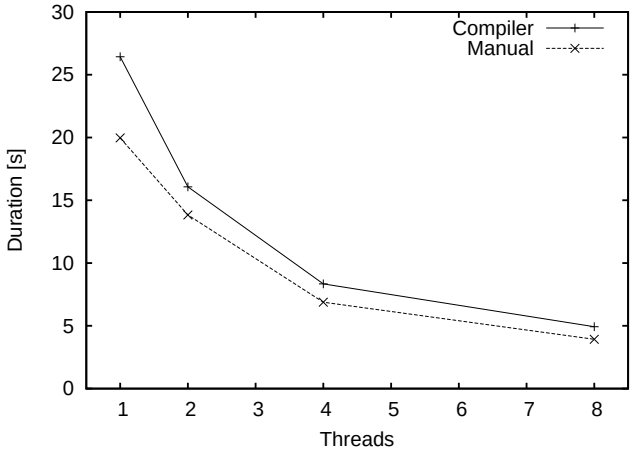


Figure 4.21: Performance of *genome* with and without STM compiler.

To do so, I extended SwissTM to support accesses to data types of sizes different than the word size. Supporting accesses to data types that are multiple words in size is simple: such an access is converted into several word-level accesses which are supported by base SwissTM. Supporting accesses to smaller data types that only occupy a part of the word, such as `char` and `short` types, is more involved. To support such data types I extend the redo log entries with a word-sized bit-mask that specifies which parts of the memory location are actually updated by the transaction. At commit time, the value is not simply copied from the redo log to the memory location. Instead, the memory location is first read, its value is then combined with the new value from the log using bitwise operators and the bit-mask, and the resulting value is stored to the memory location. Furthermore, when a write-after-write access is performed, the bit-mask is updated to correctly reflect the parts of the location that are updated by the transaction.

It is worth mentioning that most STM compilers, including the widespread GNU gcc, have adopted the Intel's ABI, which means that, in addition to the Intel's compiler, it is also possible to use SwissTM with these compilers.

When STM compiler replaces memory references in `atomic` code blocks with STM read and write calls, it typically introduces more STM calls than strictly necessary. This happens because the compiler cannot always accurately distinguish between references to transaction-local and shared data. To produce correct code, it has to, conservatively, replace all memory accesses that are not guaranteed to be transaction-local with STM reads and writes. The additional STM calls reduce the performance of the generated programs. Figure 4.21 compares performance of *genome* when using STM compiler and when manually inserting STM calls. It shows that the impact is small, but not negligible: using the STM compiler reduces performance between 20% and 30%. Importantly, the performance impact does not increase with the thread count and remains roughly equal for all thread counts. I discuss the problem of unnecessary read

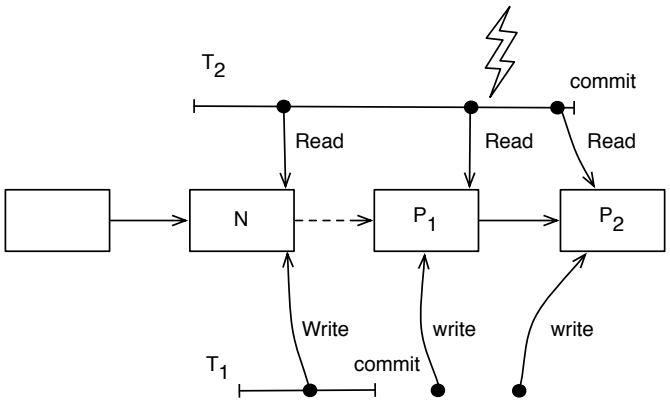


Figure 4.22: Example problem caused by use of privatization in SwissTM.

and write STM calls inserted by the compiler and its impact on performance in more detail in Section 5.5.

4.5.2 Privatization safety

The cost of executing an STM read or write call is much higher than the cost of performing the same access using an ordinary CPU instruction. Therefore, it is often beneficial for a thread to execute a short transaction and make some objects private to itself, for example by removing the object from a shared collection data structure, and then perform further processing on the objects using non-transactional accesses. This idiom is called *privatization* [106].

Base SwissTM, as well as other similar STMs that use invisible reads, such as TinySTM, TL2, and McRT-STM, does not support privatization idiom. Figure 4.22 shows an example execution where transactional reads observe the effects of non-transactional writes to privatized data. In the figure, two threads access a shared linked list. Transaction T_1 privatizes list nodes P_1 and P_2 by setting the forward pointer of node N, which precedes P_1, to `NULL`. After T_1 commits, its thread proceeds to non-transactionally modify the privatized nodes. Transaction T_2 traverses the list concurrently with T_1. Before T_1 commits, it gets a reference to P_1 by reading the forward pointer of N. Once T_1 commits, SwissTM algorithm guarantees that T_2 will fail its next read-set validation and will thus abort. However, T_2 continues to execute without validating, as none of the non-transactional writes performed by the other thread updates the *orecs* corresponding to P_1 and P_2. Therefore T_2 observes some of the non-transactional writes performed by the other thread. In case T_2 is a read-only transaction, it will even be allowed to commit, as read-only transactions do not perform validation at commit time. Similar problems occur even when transactions incrementally validate their read-sets on every read, and with STMs that use in-place updates or non-blocking designs [107].

To make SwissTM privatization-safe, all non-transactional accesses to the privatized data need to be postponed until the other threads in the system observe the updates performed

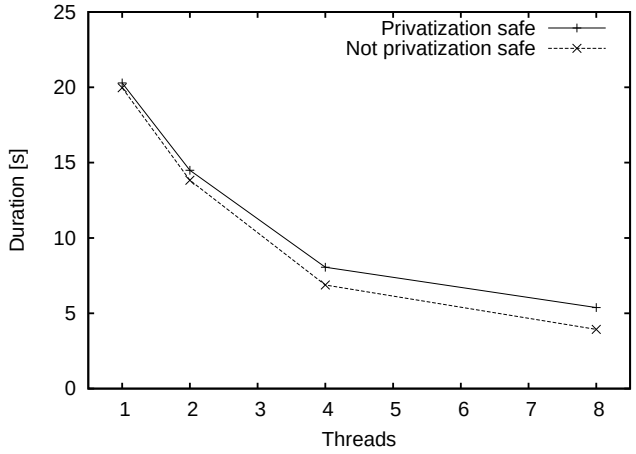


Figure 4.23: Performance of *genome* with and without privatization support.

by the privatizing transaction. In the example above, this means that non-transactional writes to P_1 and P_2 would be postponed until T_2 validates or finishes. To support correct privatization of data in SwissTM, I implemented a privatization barrier which needs to be executed by the thread that executes the privatizing transaction after the transaction commits, but before accessing the privatized data non-transactionally. To provide more general support for privatization, the barrier can be executed after each transaction commit, as a part of the STM commit call, regardless of whether the transaction privatizes any data or not. Such *transparent* support for the privatization idiom is considered by many as necessary in a complete STM system [24, 69, 85]. Inside the barrier, the thread ensures that each transaction concurrent to the privatizing transaction either completes or its read-set validity timestamp becomes equal to or higher than the privatizing transaction's commit timestamp, before continuing execution.

Figure 4.23 illustrates the impact of threads executing the privatization barrier on performance of *genome*. The experiment demonstrates the worst-case cost of privatization, as threads execute the privatization barrier after each transaction to transparently ensure privatization safety, although none of the transactions privatize data. The figure shows that the performance impact increases with the number of threads: with a single thread it is barely noticeable at around 1%, but it keeps increasing and reaches 35% with eight threads. I return to the privatization problem and discuss the impact of providing privatization support on performance of SwissTM in more detail in Section 5.6.

It is interesting to note that the privatization problem is similar to the memory management problem discussed in Section 4.2.4. Actually, memory management is an instance of privatization: before deallocating an object, the thread first uses a transaction to make the object inaccessible to other threads, hence privatizing it, and then it passes the object to the underlying non-transactional memory allocator that performs non-transactional accesses to the object. The major difference between memory management and the general privatization

problem is that memory deallocations can be buffered and performed at a later time, to improve performance as described, whereas that is not possible in general case of privatization. This makes it much more expensive to solve the general privatization problem than the memory deallocation problem.

4.6 Summary

This chapter presents SwissTM, an effective compilation of STM design choices for mixed workloads characterized by non-uniform, dynamic data structures and varying transaction sizes. Those kinds of workloads are inherent to many applications that might be expected to significantly benefit from the STM paradigm and multi-core architectures. SwissTM greatly outperforms state-of-the-art STMs in precisely such workloads, while also delivering good performance in smaller-scale scenarios. It is worth pointing out that, despite the wide performance evaluation I performed, it is possible to come up with workloads in which SwissTM does not outperform other STMs, particularly STMs that are optimized for precisely those workloads.

Not surprisingly, the design of SwissTM is a result of trial-and-error. I reported on various choices that might have seemed natural at some point, but revealed inappropriate in certain workloads, such as the use of the Greedy contention manager or the pure eager conflict detection. Besides those, I also experimented with nested transactions, using closed nesting, and multi-versioning schemes, but I could not see a clear advantage of those techniques in the considered workloads.

5 Practical STM Performance

The evaluation presented in the previous chapter demonstrates that SwissTM indeed performs well compared to other state-of-the-art STMs across a wide range of workloads. In all cases it either outperforms them, sometimes significantly or matches their performance. The evaluation, however, fails to provide insight into several very important questions: How does SwissTM, and STM in general, compare to other synchronization techniques? Could STM be used in practice to outperform locking and lock-free synchronization, or, at least, sequential single-threaded code, or are the overheads of book-keeping in software simply too high for STM to be usable in practice?

Perhaps surprisingly, this question attracted little attention by others. The only real analysis of STM's suitability for use in practical systems concluded that STM is not more than a "research toy" [19], as it failed to outperform sequential, single-threaded code in most cases, even when using eight hardware threads. My motivation in this chapter is to revisit these conclusions, by investigating whether STM can be used in practice, at least for some workloads, or is it really just a "research toy". To this end, I conducted a similar, but more thorough, analysis of the performance of SwissTM and its comparison to sequential code. I used a wider range of benchmarks, including the ones from previous chapter, and two computer systems that support higher levels of concurrency. The evaluation also considered several STM programming models, with different combinations of STM compiler support and support for the privatization idiom, which enables me to discuss the best STM programming model from the point of view of both usability and performance.

5.1 Overview

Whereas a well-optimized STM implementation can certainly reduce the overheads of book-keeping in software, as shown in the previous chapter, the STM overheads still remain rather high. A complete STM system usually includes an STM compiler and also supports the privatization idiom in some form [5, 85]. Therefore, there are three main sources of overheads in STM:

1. **Synchronization costs.** With STM, each access to a shared memory location from a transaction is performed by an STM read or write call. In contrast, with sequential code, these accesses are performed by a single CPU instruction. As the previous chapter illustrates, STM read and write routines are significantly more expensive than corresponding CPU instructions because they maintain book-keeping data about the accesses they perform. With SwissTM, each access: (1) checks for a previous write to the same location, (2) detects conflicts with other concurrent transactions, (3) updates its read- or write-set, (4) validates the read-set, and also (5) logs the new value in the write log, when performing a write access. Therefore, even the fast path of the read and write calls consists of more than 10 instructions, making them significantly more expensive to execute than ordinary CPU read and write instructions. Furthermore, some of these steps also use expensive atomic instructions, such as *compare-and-swap* for acquiring the locks, or access shared meta-data, which increases the cache miss rate. This further increases the overheads of SwissTM's reads and writes. Similarly to SwissTM, other STMs perform analogous steps and their reads and writes have comparable costs. All of these overheads significantly reduce single-threaded performance when compared to sequential code.

2. **Compiler over-instrumentation.** To use STM, STM library calls for starting and committing transactions need to be inserted in the code and all memory accesses from inside transactions have to be replaced by STM read and write calls. This process is called *instrumentation*. The instrumentation of a program can be *manual*, when programmers modify the code manually, as was assumed in previous chapters. To truly make the programming with STM simple, however, an STM compiler [45, 57, 85] is needed, in which case it is said that the instrumentation is *compiler-based*. With the compiler, the programmers only need to specify which sequences of statements have to be executed atomically, by enclosing them in atomic blocks, as the examples in Chapter 2 show. The compiler then generates code that invokes appropriate STM read and write calls. While using an STM compiler significantly reduces programming complexity, it can degrade the performance of resulting programs, when compared to manual instrumentation, due to *over-instrumentation* [19, 38, 121]. As discussed in the previous chapter, the compiler cannot always precisely distinguish between accesses to shared and transaction-local data and, hence, it has to instrument the code conservatively. Such conservative instrumentation can result in unnecessary calls to STM, when STM is used to access transaction-local data, further increasing the overheads of using STM.

3. **Transparent privatization.** Making some shared objects private to a thread is known as the privatization idiom. Privatization is typically used to allow non-transactional accesses to some data, either to improve performance by avoiding costs of STM calls when accessing private data or to support legacy code. As privatization is not supported by the base STM algorithms that use invisible reads, privatizing data with such STMs can lead to various race conditions [106]. Invisible-read STMs can be extended to provide support for safely privatizing data, as I described in the context of SwissTM

	Instrumentation	Privatization
SwissTM-ME	manual	explicit
SwissTM-CE	compiler	explicit
SwissTM-MT	manual	transparent
SwissTM-CT	compiler	transparent

Table 5.1: Considered programming models.

in Section 4.5.2. Supporting privatization, however, introduces additional overheads, due to the necessary synchronization among the threads. There are two different approaches to enabling privatization of data: (1) a programmer marks transactions that privatize data, so the STM can safely privatize data only for these transactions, or (2) the STM ensures that all transactions safely privatize data. The first approach is called *explicit* and the second *transparent*. Explicit privatization places additional burden on programmers, but it does not incur the privatization overheads when they are not necessary. On the other hand, transparent privatization makes programming simpler, but it incurs runtime overheads that can be high [121]. In an extreme case when transactions do not privatize any data, explicit approach to privatization incurs no cost, while transparent privatization impacts all transactions significantly. In this case, transparent privatization is particularly expensive, as threads do not benefit from being able to access some shared data non-transactionally, but pay the costs of privatization nevertheless. Many believe that it is crucial for a complete STM system to provide transparent privatization support, as it makes the exposed programming model simpler and easier to understand [24, 69, 85].

The experiments from the previous chapters, as well as many research papers have conveyed the scalability of a number of STMs with the increasing number of threads on various benchmarks [2, 3, 8, 17, 28–30, 43, 54, 62, 69, 76, 77, 85, 92, 93, 99, 102, 108], demonstrating that STMs can indeed scale well. These results, however, do not compare STM to sequential code, thus completely ignoring the fundamental question of whether STM can be a viable option for actually speeding up the execution of programs, or are the various overheads simply too high and prevent any practical use of STM.

Two notable exceptions are the work on STAMP benchmarks [17] and evaluation performed by Cascaval *et al.* [19]. The STAMP evaluation shows that STM outperforms sequential code in most STAMP benchmarks, but it uses experiments based on a hardware simulator. Recently, the evaluation by Cascaval *et al.* [19] shows that, with real hardware, STM performs worse than sequential code even when using up to eight hardware threads, and for that reason it is argued that STM is only a "research toy". These findings are based on the experiments with micro-benchmarks and a subset of the STAMP benchmark suite with specific configurations.

In this chapter, I revisit the conclusions by Cascaval *et al.* [19] by presenting an extensive comparison of SwissTM performance to performance of sequential code. My experiments

	SwissTM-ME			SwissTM-CE			SwissTM-MT			SwissTM-CT		
System	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
SPARC	9.1	1.4	29.7	-	-	-	5.6	1.2	23.6	-	-	-
x86	3.4	0.54	9.4	3.1	0.8	9.3	1.8	0.34	5.2	1.7	0.5	5.3

Table 5.2: Summary of SwissTM speedup over sequential code.

were based on a larger set of benchmarks and real hardware that supports much higher levels of concurrency than in the previous chapters and the evaluation of Cascaval *et al.* Besides the benchmarks from Chapter 4, I used three additional micro-benchmarks, based on the linked-list, skip-list, and hash-table data structures. In the experiments, I used 17 workloads in total, which span a wide range of workload characteristics. The experiments were conducted on two hardware platforms: a Sun Microsystems UltraSPARC T2 CPU system, referred to as *SPARC* in the remainder of the chapter, which supports 64 hardware threads, and a four quad-core AMD Opteron x86 CPU system, referred to as *x86* in the remainder of the chapter, which supports 16 hardware threads. These experiments constitute the most extensive performance comparison of STM to sequential code to date, both in terms of used benchmarks and hardware architectures. The goal of the experiments is to determine whether SwissTM can outperform sequential code and, thus, promise to speed up actual real-world code in the near future. The conclusions based on the collected results, however, do not apply just to SwissTM and they evaluate the ability of STM in general to achieve good performance.

To be exhaustive, I considered all combinations of privatization and compiler support for STM, summarized in Table 5.1. I used the techniques described in Section 4.5.1 to integrate SwissTM with Intel's STM compiler [85] and in Section 4.5.2 to make SwissTM privatization-safe.

Table 5.2 summarizes the results of the experiments. It illustrates that SwissTM-ME outperforms sequential code on both systems and on all benchmarks, except for high contention write-dominated STMBench7 workload on *x86*. The achieved speedups are in some cases impressive: SwissTM outperforms sequential code by up to 29x on *SPARC* with 64 concurrent threads and by up to 9x on *x86* with 16 concurrent threads. The experiments confirm that, while compiler over-instrumentation impacts the performance, it does not significantly impact the scalability. SwissTM-CE outperforms sequential code in all STAMP benchmarks with high contention and in all but one micro-benchmark out of fourteen workloads.[1] The speedups are slightly lower than without the compiler, but they are still high, as SwissTM-CE outperforms sequential code by up to 9x with 16 concurrent threads on *x86*. On the other hand, supporting transparent privatization impacts both the performance and the scalability of SwissTM, sometimes even significantly. However, SwissTM-MT still outperforms sequential code in all benchmarks on *SPARC* and in all but several high-contention workloads on *x86*, outperforming sequential code on 14 out of 17 workloads on *x86*. The speedups achieved

[1] I did not collect the measurements for SwissTM-CE on the *SPARC* system, as I did not have a compiler for it at my disposal. Furthermore, Intel's STM compiler does not compile STMBench7 correctly, so I was not able to collect the STMBench7 results on *x86* either.

with SwissTM-MT are lower than with SwissTM-ME: it outperforms sequential code by up to 23x on *SPARC* with 64 concurrent threads and by up to 5x on *x86* with 16 threads, compared to 29x and 9x respectively with SwissTM-ME. Furthermore, SwissTM performs reasonably well in several workloads even when using STM compiler to instrument the benchmarks and providing support for privatization transparently: SwissTM-CT outperforms sequential code on all but two high-contention STAMP benchmarks and two micro-benchmarks, meaning it outperforms sequential code on 10 out of 14 workloads on *x86*, by up to 5x with 16 concurrent threads.

To summarize, the performed experiments show that SwissTM indeed outperforms sequential code in most configurations and benchmarks, offering already now a viable paradigm for concurrent programming. These results are important as they support initial hopes about the good performance of STM, and motivate further research in the field. The results also support reasoning about the best STM programming model and they show that, at the moment, STM-CE hits the "sweet spot" considering both programmability and performance, making it best suited for practical use today.

The results of my experiments contradict these of Cascaval *et al.* [19] and I believe that reasons for this are three-fold: (1) in STAMP workloads by Cascaval *et al.* threads encounter higher contention than in the default STAMP workloads, which I used in my experiments, (2) I used different hardware systems with higher levels of parallelism, and (3) I used SwissTM, which has higher performance than STMs used in the evaluation performed by Cascaval *et al.* according to the experiments presented in the previous chapter.

5.2 Experimental settings

The experimental settings used in this chapter are similar to the settings in the previous chapters. The biggest difference is the use of larger machines that support more hardware threads, and the use of three additional micro-benchmarks.

Hardware. I used the following system configurations for the experiments:

- A system with a Sun Microsystems UltraSPARC T2 CPU at 1.2 GHz and 32 GB memory running Solaris 10 operating system. The CPU has 8 cores, where each core multiplexes 8 hardware threads, for a total of 64 supported hardware threads. For brevity, the system is referred to as *SPARC*.

- A system with four quad-core AMD Opteron CPUs at 2.2 GHz and 8 GB memory running Linux operating system with kernel version 2.6.22.19. This system supports a total of 16 hardware threads. For brevity, the system is referred to as *x86*.

Benchmarks. I used the same benchmarks to compare the performance of SwissTM and sequential code as in the previous chapter: STMBench7 to evaluate SwissTM on large-scale workloads, STAMP to evaluate it on a wide range of medium-scale workloads, and micro-benchmarks to measure performance with small transactions. I used four micro-benchmarks, based on the red-black tree, linked-list, skip-list, and hash-table data structures. All benchmarks have the same structure as the red-black tree described in the previous chapter, except that they use different data structures to implemented the shared integer set. In the experiments, I configured them to use the range of 131,072 values and a mix of 90% lookup, 5% insert, and 5% remove operations.

On *x86*, all benchmarks, including the manually instrumented ones, were compiled using the Intel's experimental compiler, which supports STM language constructs. For compiler-instrumented benchmarks, I used language constructs, such as `tm_pure` and `tm_waiver`, to turn off STM instrumentation of code blocks where safe, as is also done in the original STAMP implementation which uses manual instrumentation. On *SPARC*, I used the available GNU gcc compiler, which did not support transactional constructs. As I did not have an STM compiler for the SPARC architecture at my disposal, I could not perform experiments with the compiler-instrumented benchmarks on *SPARC*. Likewise, I could not collect the measurements for compiler-instrumented STMBench7 on *x86* because of runtime errors in the compiler-instrumented STMBench7, which were the result of bugs in the compiler.[2]

Experimental methodology. The execution of each experiment was repeated for at least five times, to reduce the variance in the collected measurements. The graphs report averages from these executions. SwissTM performance is reported as the speedup over the sequential code, where the speedups are calculated as follows: $Speedup = Throughput_{SwissTM} \big/ Throughput_{Sequential}$ for STMBench7 and micro-benchmarks, and $Speedup = Duration_{Sequential} \big/ Duration_{SwissTM}$ for STAMP.

5.3 SwissTM-ME performance

First, I present the experiments with SwissTM-ME, which uses manual instrumentation and supports privatization through explicit calls. As none of the benchmarks uses privatization, the threads do not execute privatization barriers at all, meaning that no overheads related to privatization are incurred in these experiments.

SPARC. Figure 5.1 depicts SwissTM-ME performance on *SPARC*. It shows speedup of code that uses SwissTM-ME with different thread counts, over sequential, non-instrumented code. The results convey that SwissTM-ME outperforms sequential code already with a small number of threads: with just four threads it outperforms sequential code on 14 out of 17 work-

[2] Here, as in Chapter 3, STMBench7 demonstrated its usefulness as a correctness testing tool.

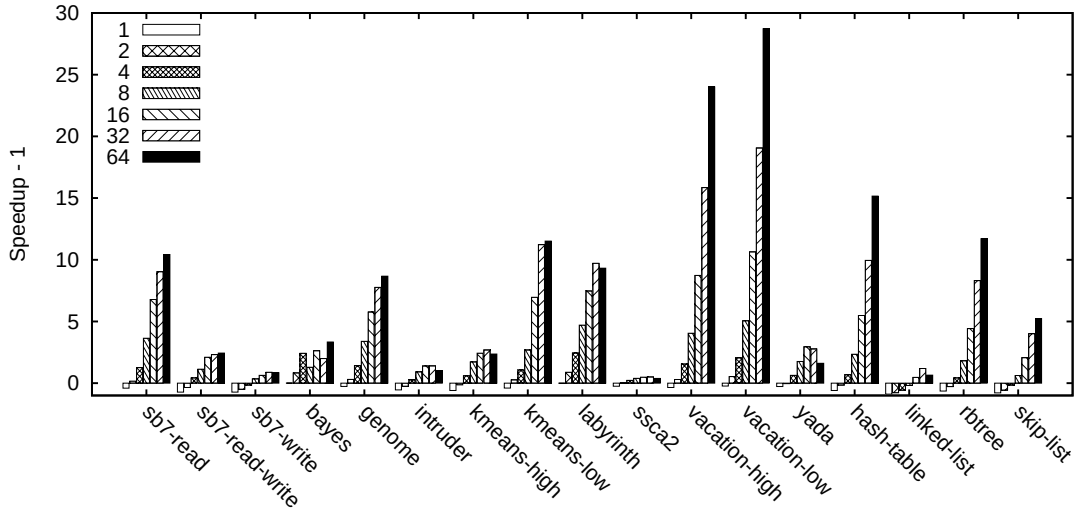


Figure 5.1: SwissTM-ME on *SPARC*.

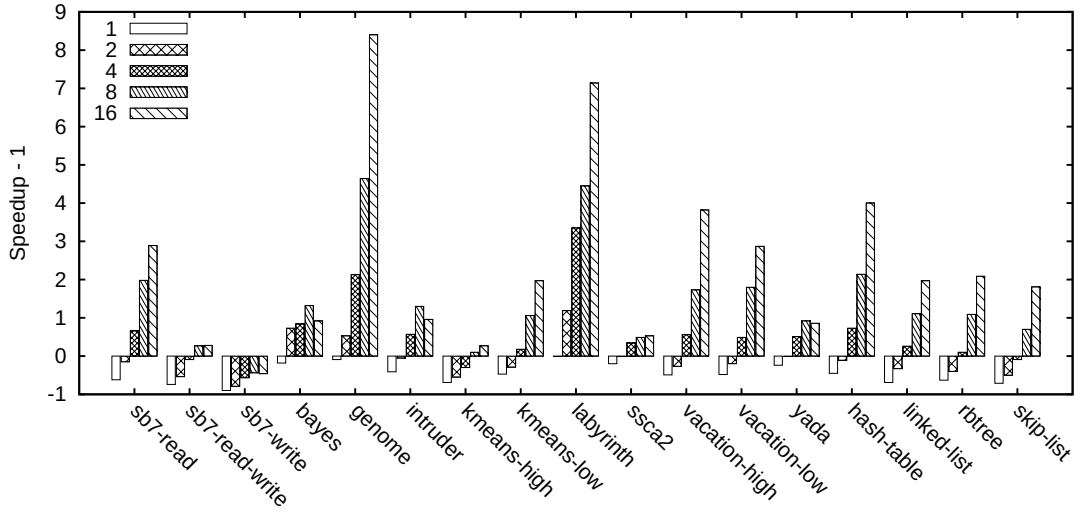


Figure 5.2: SwissTM-ME on *x86*.

loads. This illustrates that SwissTM-ME is indeed a viable option for writing parallel code performance-wise, as it can achieve reasonable performance on today's systems with just a handful of threads. Furthermore, the figure shows that SwissTM-ME outperforms sequential code on all used workloads when using all 64 threads supported by the system, and that it achieves very good speedups in several cases. Performance is the best on *vacation-low* workload, where SwissTM-ME is 29x better than the sequential code. In several other cases the speedups are only slightly lower, as SwissTM-ME outperforms sequential code by more than 10x on the following six workloads: *sb7-read*, *kmeans-low*, *labyrinth*, *vacation-high*, *hash-table*, and *rbtree*.

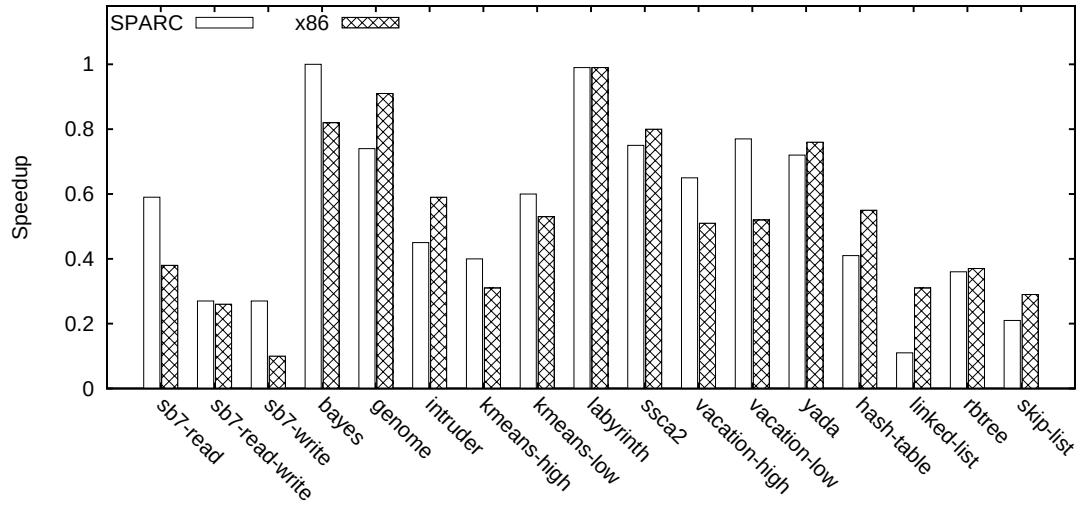


Figure 5.3: SwissTM-ME single thread overheads.

Despite rather good speedups over sequential code in several cases, SwissTM-ME does not always outperform sequential code by large margins. For example, on the *sb7-write*, *ssca2*, and *linked-list* workloads the speedups are lower than 2x even when using 64 threads, mostly due to high contention encountered by threads in these workloads. In general, the experiments confirm that the less contention the workload exhibits, the more benefit can be expected from STM. For example, SwissTM outperforms sequential code by more than 11x on low-contention *sb7-read*, but by less than 2x on high-contention *sb7-write* workload of the same benchmark.

x86. Figure 5.2 presents the performance of SwissTM-ME on the *x86* system. The performance is comparable to the performance on *SPARC*, as SwissTM-ME outperforms sequential code when using four threads on 13 out of 17 workloads. Furthermore, when using all 16 threads supported by the system, SwissTM clearly outperforms sequential code on all workloads except on the challenging, high-contention *sb7-write* workload. The performance gain, when compared to sequential code, is lower than on *SPARC*: the speedups are up to 9x on *x86* compared to up to 29x on *SPARC*, mostly because there are also fewer hardware threads on *x86*. Also, the relative performance is better on *SPARC* because all threads execute on the same chip which reduces the costs of inter-thread communication compared to multi-chip *x86*, and because *SPARC* has much lower performance.

Singe-thread overheads. It is interesting to compare the performance of single-threaded SwissTM-ME to sequential, non thread-safe, code. Such a comparison illustrates "pure" bookkeeping overheads of SwissTM without considering overheads introduced by the compiler and transparent support for privatization. Figure 5.3 depicts the speedup of STM code over the sequential code with a single thread. It shows that the overheads with a single thread

vary significantly across the benchmarks, with performance ranging from just 10% of the performance of sequential code, for example on *linked-list* on *SPARC* and *sb7-write* on *x86*, to virtually the same performance as sequential code, for example on *labyrinth* on both systems. On average, SwissTM-ME with a single thread performs almost twice as slow as the sequential code. This means that it needs at least two threads on average to achieve the performance of the sequential code, even with perfect scalability. As programs rarely scale perfectly, SwissTM-ME actually needs to use more than two threads to do so in most cases: with two threads, it outperforms sequential code in just 11 out of 34 experiments on both systems combined.

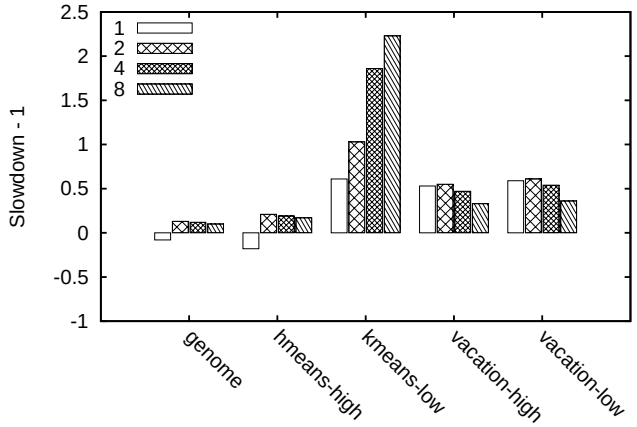
Summary. To summarize, SwissTM-ME achieves good performance on both *SPARC* and *x86* systems, clearly showing that STM-ME algorithms can scale and perform well in various settings. It is, however, important to point out that, while SwissTM-ME outperforms sequential code in all but one workload, the achieved speedups are not very impressive in all cases, such as, for example, 1.4x speedup with 64 threads on *ssca2* on *SPARC*, or 1.3x speedup with 16 threads on *kmeans-high* on *x86*. These and similar examples confirm that STM, while showing great promise for certain workloads, is not the prefect fit for all of them.

Further optimizations. On some of the used workloads, such as, for example, *intruder* and *yada*, performance degrades when too many threads are used, due to increased contention among threads. A possible approach to solving this issue is to modify the thread scheduler and have it avoid running more concurrent threads than is optimal for a given workload. For this, the STM runtime would have to provide some additional information, and it is likely that such a solution would not be easily applicable to all programs.

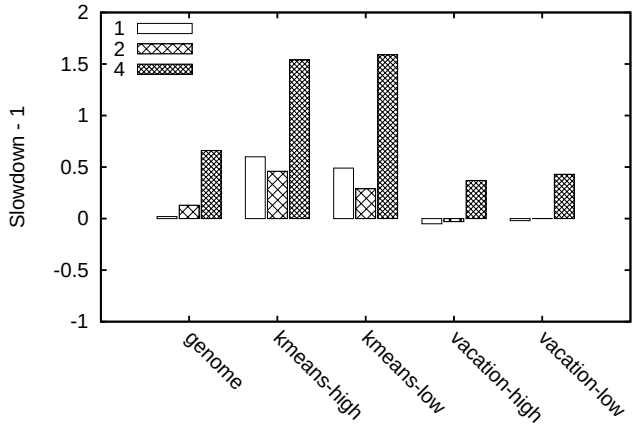
5.4 Contradicting earlier results

In contrast to the results presented above, the experiments performed by Cascaval *et al.* [19] indicated that STMs do not perform well on three of the STAMP benchmarks I also used: *kmeans*, *vacation*, and *genome*. In their experiments, the used STMs failed to achieve the performance of sequential code both on *kmeans* and *vacation* benchmarks, whereas the performance on *genome* was only about 2.5x better than sequential, even when using eight threads. In contrast, in my experiments, SwissTM-ME outperforms sequential code on all of those benchmarks when using eight threads on *x86*, achieving 5.6x speedup on *genome*. In the following, the three main reasons for this dramatic difference in the observed STM performance are discussed:

1. *Workload characteristics.* A closer look at the experimental settings of Cascaval *et al.* reveals that the STAMP benchmarks were configured in a specific way, which increased the contention among threads. In contrast, I used the default STAMP workloads, as defined in STAMP distribution 0.9.9. The results presented in Figures 5.1 and 5.2 illustrate



(a) Workload impact



(b) Hyper-threading impact

Figure 5.4: Impact of experimental settings used by Cascaval *et al.* [19] on STM performance.

the well known fact that the performance of parallel programs is typically the lowest in highly contended workloads, which means that the experiments performed by Cascaval *et al.* actually evaluated STM in particularly unfavorable settings.

To evaluate the impact of the workload characteristics on the performance of SwissTM-ME, I repeated the experiments using both default STAMP workloads and the workloads used by Cascaval *et al.* on a system with two quad-core Xeon CPUs, with 8 hardware threads in total. I used a different machine than in the previous experiments because this one is more similar to the machine used by Cascaval *et al.* Figure 5.4(a) depicts the slowdown of the higher contended workload compared to the default STAMP workloads, calculated as: $Slowdown = Speedup_{Default} \Big/ Speedup_{HighCont}$. I used both the low-contention and high-contention default workloads for *kmeans* and *vacation* benchmarks to understand the performance difference in both cases. The figure shows that the custom workload configuration indeed degrades the performance of SwissTM-ME.

The biggest difference is on the *kmeans-low* workload, with the slowdown of around 3.2x. The performance impact is also significant on the *kmeans-high* workload, where it is around 20%, and on both *vacation* workloads, where it is around 35%. The performance impact is the smallest with *genome*, but it is still non-negligible at around 10%.

2. *Different hardware.* The experiments presented in this chapter use systems which support more hardware threads than the ones in experiments of Cascaval *et al*: 64 and 16 threads compared to only 8. This naturally benefits parallel code as using more threads typically leads to better overall performance.

   Furthermore, the *x86* system does not use hyper-threading, whereas the one used by Cascaval *et al.* does, which also impacts the performance, as hardware thread multiplexing sometimes hampers performance. To better understand the impact of the hyper-threading, I repeated the experiments with the default STAMP workloads on a system with two single-core hyper-threaded Xeon CPUs, for a total of 4 hardware threads. Figure 5.4(b) depicts the slowdown on this system compared to a similar machine without hyper-threading. The slowdown is calculated using speedups over sequential code on different systems: $Slowdown = Speedup_{NoHyper} \Big/ Speedup_{Hyper}$. The figure shows that hyper-threading impacts performance significantly, especially with higher thread counts. This results in a slowdown of around 65% for the *genome* workload and around 40% in the two *vacation* workloads. The performance difference in *kmeans* workloads is significant even with a single thread, which is the result of differences in the used CPUs that are not related to hyper-threading. Still, even with *kmeans*, slowdown with four threads is much higher than with one and two threads, meaning that hyper-threading impacts scalability adversely in *kmeans* workloads too.

3. *More efficient STM.* A part of the difference in the obtained results stems from my use of a more efficient STM. I did not directly compare SwissTM to all STMs in the evaluation of Cascaval *et al.* because (1) IBM STM is not freely available, so I could not use it in the experiments, and (2) McRT-STM [85, 99] can only be used with the compiler instrumentation and transparent privatization, which does not allow for a fair comparison with SwissTM-ME. However, the results presented in the previous chapter show that SwissTM outperforms TL2 on the STAMP benchmarks used by Cascaval *et al*, and TL2 performed comparably to the other STMs in their experiments. Therefore, it can be concluded that a part of the difference in the observed STM performance indeed stems from the use of SwissTM in my experiments, instead of other STMs with lower performance.

In addition to the presented results, I have confirmed that STM in general can achieve better performance than was suggested by Cascaval *et al*, by repeating a subset of the experiments with TL2 [28], McRT-STM [85, 99], and TinySTM [43]. The performance results with the Bartok STM [57], on a subset of STAMP benchmarks, were also provided to us and they too confirm the general conclusions from above. These results are presented in a separate technical report [32].

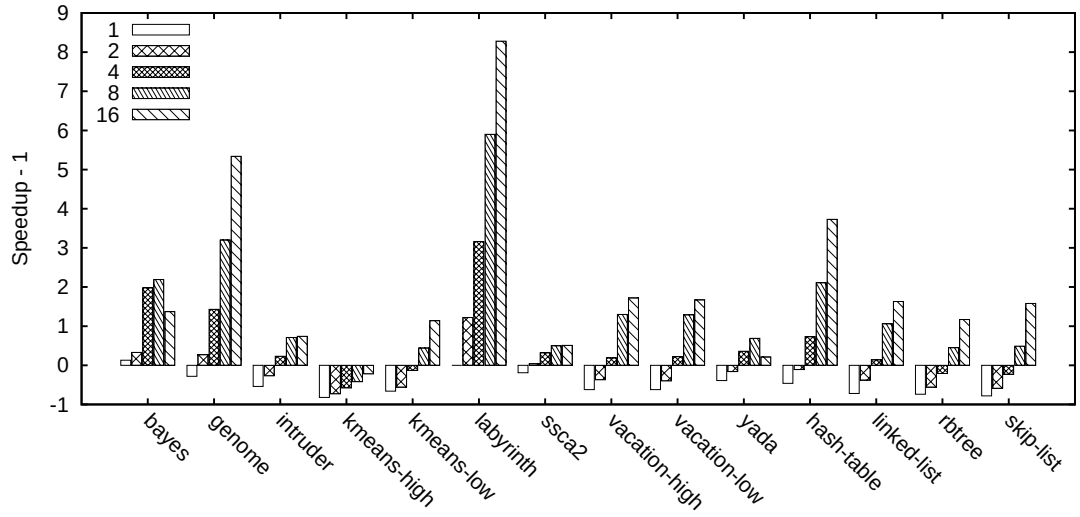


Figure 5.5: SwissTM-CE on *x86*.

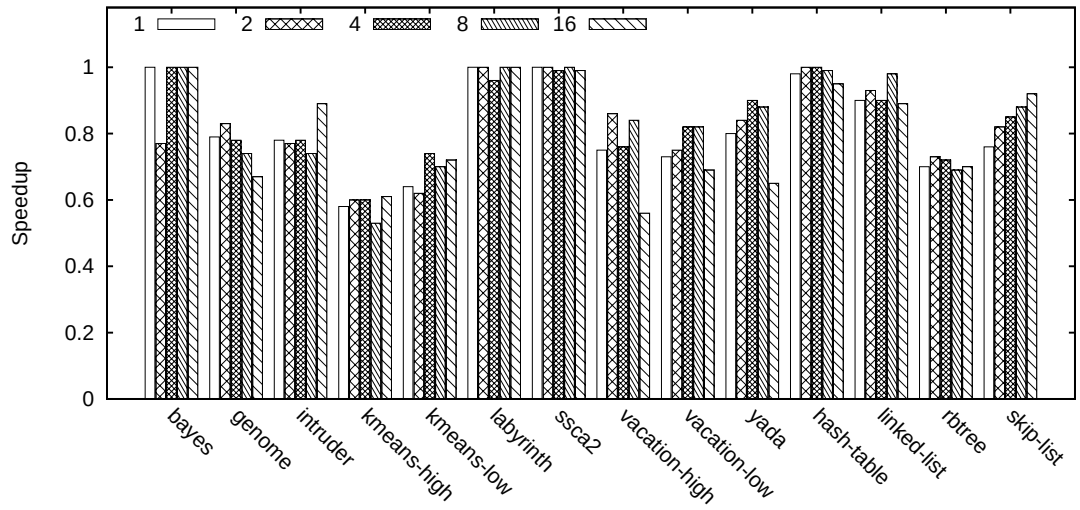


Figure 5.6: Compiler over-instrumentation overheads on *x86*.

5.5 SwissTM-CE performance

Next, I evaluate SwissTM-CE, which uses Intel's STM compiler to instrument the benchmark code and relies on programmers' hints to provide explicit privatization support. As discussed, compiler instrumentation often replaces more memory references by STM calls for reading and writing data than strictly necessary, resulting in the reduced performance of generated code. Ideally, the compiler would replace memory accesses with STM calls only when they referenced data that is shared by threads. However, the compiler does not have (1) information about all uses of variables in the whole program and (2) semantic information about variable

Threads	Min	Max	Avg
1	0	0.42	**0.16**
2	0	0.4	**0.17**
4	0	0.4	**0.11**
8	0	0.47	**0.11**
16	0	0.44	**0.17**

Table 5.3: Summary of the compiler over-instrumentation overheads on *x86*.

use, which is typically available only to the programmer. Consequently, the compiler cannot always determine which variables are read-only or private to the transaction, and thus can be accessed non-transactionally. This is why the compiler instruments the code conservatively, usually generating more STM calls than necessary. Unnecessary STM calls reduce performance because they are more expensive than the CPU instructions that they replace. It is very important to evaluate these over-instrumentation overheads because the compiler is the crucial component of the full-fledged STM system, as it enables programmers to use STM in a truly simple manner.

The speedups of SwissTM-CE when running with different numbers of threads over sequential code are presented in Figure 5.5. As discussed, I could not collect the measurements on *SPARC*, for lack of STM compiler, and for STMBench7 on *x86* due to the limitations of the STM compiler I used. The figure shows that the performance is lower than the performance of SwissTM-ME, but is still rather good: SwissTM-CE outperforms the sequential code on 10 out of 14 workloads with four threads and on all but 1 workload with eight threads. Similarly to the results with SwissTM-ME, the highest observed speedup is over 9x, on the *labyrinth* workload. Overall, SwissTM-CE outperforms sequential code in all benchmarks but *kmeans-high*. However, it scales well on *kmeans-high* promising to outperform the sequential code on systems with more hardware threads.

The overheads of using an STM compiler with SwissTM are illustrated in Figure 5.6 and Table 5.3, by comparing the performance of SwissTM-ME and SwissTM-CE. The figure depicts the speedup of SwissTM-CE over SwissTM-ME calculated as $Speedup = Speedup_{SwissTM\text{-}CE} / Speedup_{SwissTM\text{-}ME}$, using the speedups of the two SwissTM variants over the sequential code. The table shows the overheads calculated as $1 - Speedup$. The over-instrumentation costs are typically not very high: they remain around 20–30% for all workloads but *kmeans* where they are about 40%. Furthermore, in several workloads, such as *labyrinth*, *ssca2*, and *hash-table*, the performance is only barely impacted by over-instrumentation, resulting in similar performance of SwissTM-CE and SwissTM-ME. Interestingly, the over-instrumentation overheads remain approximately the same for all thread counts, conveying that over-instrumentation does not greatly impact scalability of STM.

It is worth pointing out that the Intel's STM compiler I used already implements several optimizations to reduce the effects of over-instrumentations [38, 85]. Therefore, the overheads are not as high as they might be with a compiler that relies on less sophisticated techniques.

To summarize, the additional overheads introduced by compiler over-instrumentation remain acceptable as SwissTM-CE outperforms sequential code on 10 out of 14 workloads with only four threads and in all but 1 workload overall, and they do not impact scalability of STM.

Further optimizations. Optimizations that replace full STM load and store calls with specialized, faster versions of the same calls, to exploit the ability of some STMs to perform, for example, fast read-after-write accesses, are described in [85]. While the Intel's compiler I used indeed supports such optimizations, I did not implement the lower cost STM barriers in SwissTM, because the deferred-update STMs do not benefit significantly from them. More sophisticated techniques, such as data structure analysis performed by the compiler to optimize the generated code, could also be used [94].

Further compiler optimizations are possible in the context of higher-level languages, such as Java and C#. Several optimizations have been proposed in the context of Java to eliminate transactional accesses to immutable data and data allocated inside current transaction [3]. Data-flow analysis can also be used to eliminate some of the unnecessary transactional accesses and replace them with non-transactional ones, as in [39]. Bartok-STM [57] uses flow-sensitive inter-procedural compiler analysis, as well as runtime log filtering, to identify objects allocated in the current transaction and eliminate transactional accesses to them.

5.6 SwissTM-MT performance

Next, I turn to evaluating the performance of SwissTM variants that transparently support the privatization idiom, by measuring the performance of SwissTM-MT. Similarly to the overheads of the compiler over-instrumentation, it is very important to evaluate the overheads of transparent privatization support, as many believe that privatization support is crucial for wider inception of STM.

Validation barriers used for ensuring privatization safety, described in Section 4.5.2, require frequent communication between all threads in the system and can thus degrade performance due to (1) the time threads spend waiting for each other and (2) the increased number of cache misses. It is already known that a similar technique significantly impacts the performance and scalability of STM in certain cases [121], which my experiments confirm.

As has already been mentioned, none of the used benchmarks requires privatization. Therefore, the worst case scenario was measured: supporting transparent privatization only incurs overheads, without the performance benefits of reading and writing privatized data outside of transactions, that could compensate for these overheads. Also, the measured performance costs are specific to the choice of privatization technique and implementation, and proposals for reducing privatization costs exist [69, 78].

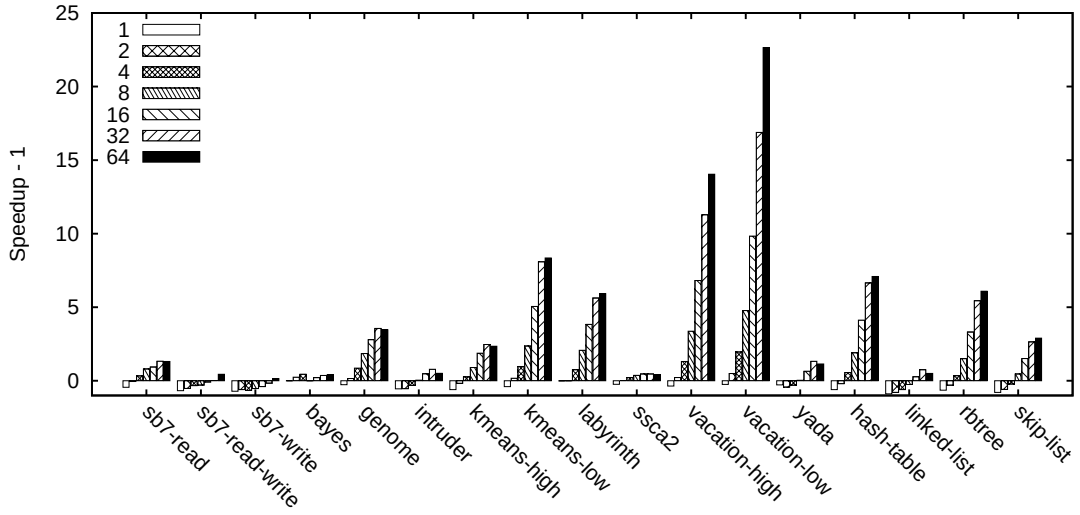


Figure 5.7: SwissTM-MT on *SPARC*.

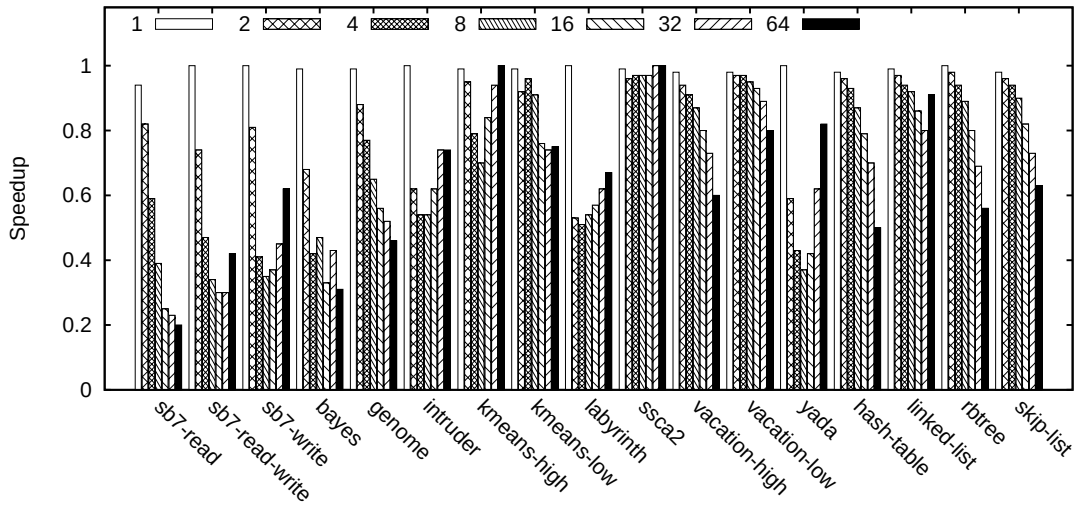


Figure 5.8: Overheads of ensuring privatization safety on *SPARC*.

SPARC. The performance of SwissTM-MT variant, which relies on manual instrumentation of benchmarks and transparently provides privatization safety, on *SPARC* is shown in Figure 5.7. The figure conveys that support for transparent privatization incurs significant overheads, but that SwissTM-MT still performs rather well, outperforming sequential code on 11 out of 17 workloads with four threads and on 13 workloads with eight threads. Similarly to SwissTM-ME on *SPARC*, SwissTM-MT outperforms sequential code in all workloads, when using all 64 threads supported by the system. The performance is, however, lower, as SwissTM-MT outperforms sequential code by up to 23x compared to 29x with SwissTM-ME, and by 5.6x on average compared to 9.1x on average with SwissTM-ME.

	SPARC			*x86*		
Threads	Min	Max	Avg	Min	Max	Avg
1	0	0.06	**0**	0	0.45	**0.08**
2	0.02	0.47	**0.16**	0.03	0.58	**0.29**
4	0.03	0.59	**0.26**	0.06	0.64	**0.4**
8	0.03	0.66	**0.32**	0.08	0.69	**0.48**
16	0	0.75	**0.35**	0.17	0.85	**0.51**
32	0	0.77	**0.34**	-	-	-
64	0	0.8	**0.35**	-	-	-

Table 5.4: Summary of transparent privatization safety overheads.

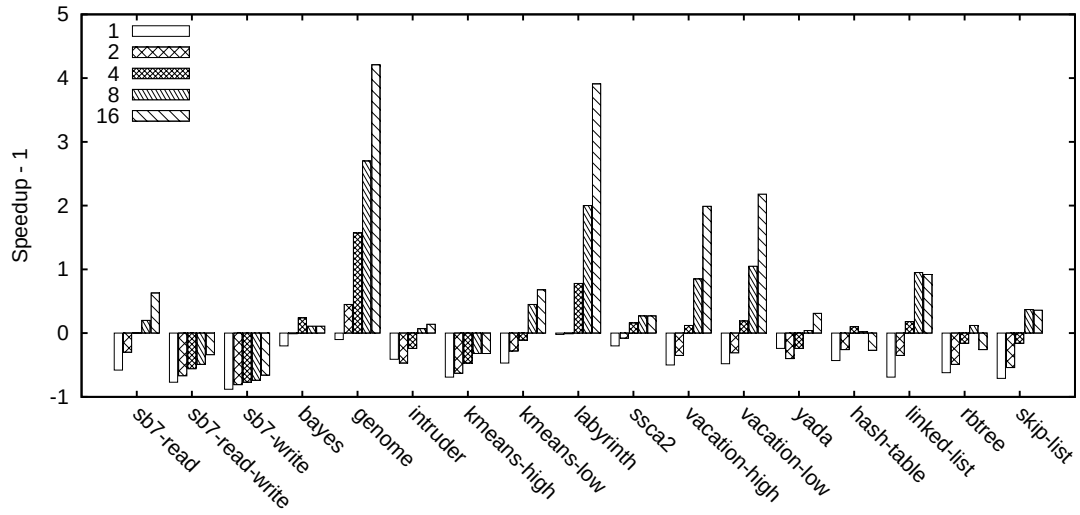


Figure 5.9: SwissTM-MT on *x86*.

Figure 5.8 and Table 5.4 convey the overheads of ensuring transparent privatization, by comparing the performance of SwissTM-MT and SwissTM-ME. The speedups and the overheads are calculated in the same way as for SwissTM-CE above. Similarly to the experiments with the compiler, transparent privatization barely impacts the performance for some of the workloads, such as, for example, *ssca2* and *kmeans-low*, whereas in some cases the privatization costs are as high as 80%, as in, for example, *sb7-read*. In contrast to the costs of complier over-instrumentation, which in the experiments remain roughly constant for all thread counts, the costs of ensuring privatization safety increase with the number of concurrent threads, thus impacting both performance and scalability of SwissTM.

x86. The results of the same experiments on *x86* are given in Figure 5.9. The data confirms the observations from the experiments on *SPARC*: the performance of SwissTM-MT is lower than the performance of SwissTM-ME, but it is still reasonably good in many cases. SwissTM-MT outperforms sequential code on 8 out of 17 workloads with four threads and on

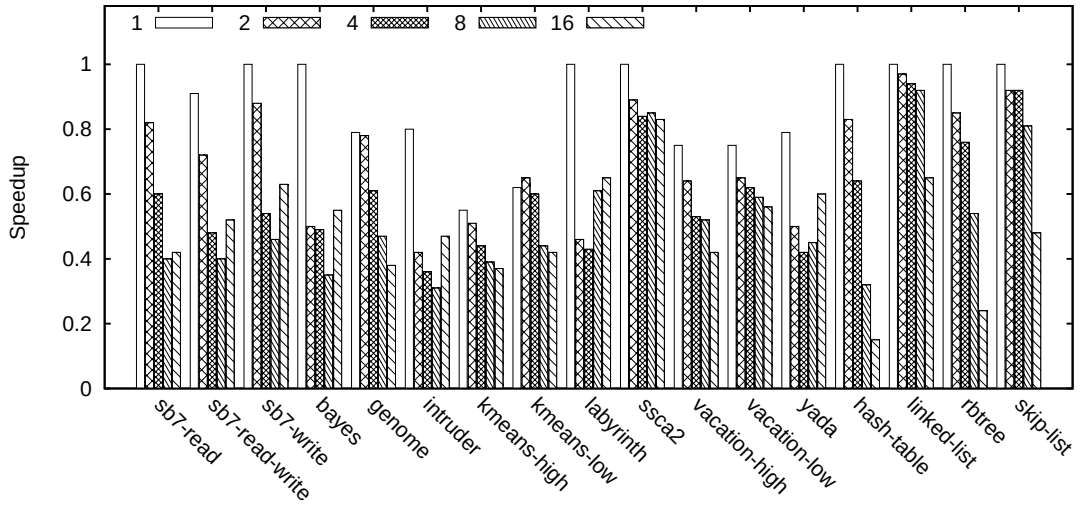


Figure 5.10: Overheads of ensuring privatization safety on *x86*.

14 workloads with eight threads. Overall, transparent privatization overheads reduce STM performance below performance of sequential code in three benchmarks: *sb7-read-write*, *sb7-write*, and *kmeans-high*. In contrast to the results on *SPARC*, the performance is impacted the most with micro-benchmarks. This is because cache contention for shared privatization meta-data induced by small transactions increases the rate of cache-misses, which are much more expensive on multi-chip *x86*, than on single-chip *SPARC*.

Figure 5.10 shows the overheads of ensuring privatization-safety transparently on *x86*, which are also summarized in Table 5.4. These are presented in the same way as the overheads of SwissTM-CE. It shows that privatization costs are sometimes as high as 80%, for example with *hash-table* and *rbtree*. In contrast to results on *SPARC*, transparent support for privatization always incurs non-negligible costs, with the smallest impact on *ssca2*, where the overheads are slightly less than 20% with 16 threads. The experiments on *x86* also confirm that the costs of transparent privatization increase with the number of threads, impairing scalability: the average cost with a single thread is about 7% and it increases to about 50% at 16 threads. Interestingly, the overheads of transparent privatization are higher than the overheads of compiler over-instrumentation with more than two threads: on average, the performance impact of privatization on *x86* with sixteen threads is almost 50%, compared to 17% impact of over-instrumentation. The impact of privatization also reaches 85% on some workloads, compared to 44% impact of over-instrumentation.

To summarize, while the impact of transparent privatization can be significant, SwissTM-MT still performs well on a wide range of workloads. Based on the presented results, it can be concluded that reducing costs of cache coherence traffic by having more cores on the same chip reduces the costs of transparent privatization, resulting in better performance

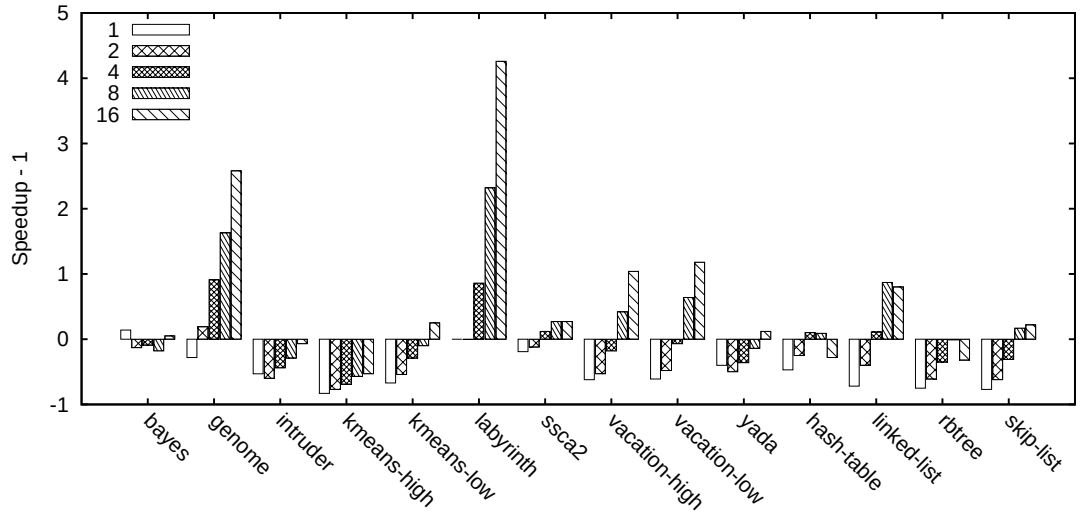


Figure 5.11: SwissTM-CT on *x86*.

and scalability. Therefore, costs of transparent privatization might be reduced on the future systems with a single processor and many cores.

Further optimizations. Two recent proposals [69, 78] aim to improve scalability of transparent privatization by employing partially visible reads. By making readers only partially visible, the cost of reads is reduced, compared to fully visible readers, and the scalability of privatization support is improved, compared to invisible readers. To implement partially visible readers, one approach uses timestamps [78], while the other proposes the use of a variant of SNZI counters [40] in STM called SkyTM [69]. In addition, SkyTM avoids using centralized privatization meta-data, which further improves scalability. Whereas these techniques improve scalability of privatization support, they greatly impact transactions that do not use privatization, and programmers that are prepared to explicitly mark privatizing transactions.

5.7 SwissTM-CT performance

I also measured the performance of SwissTM-CT, where benchmarks are instrumented by the compiler and privatization safety is ensured transparently. The results for only a subset of workloads on *x86* are available, for lack of full compiler support, as discussed. The performance of SwissTM-CT is significantly lower than of any other SwissTM variant, but still SwissTM-CT outperforms the sequential code in all workloads except *intruder*, *kmeans-high*, *hash-table*, and *rbtree*. However, it requires higher thread counts to outperform sequential code than the other SwissTM variants: with two threads SwissTM-CT performs better than sequential code only on the *genome* and *labyrinth* workloads of all 14 workloads used. With four threads STM-CT outperforms sequential code on 5 workloads, and with eight threads

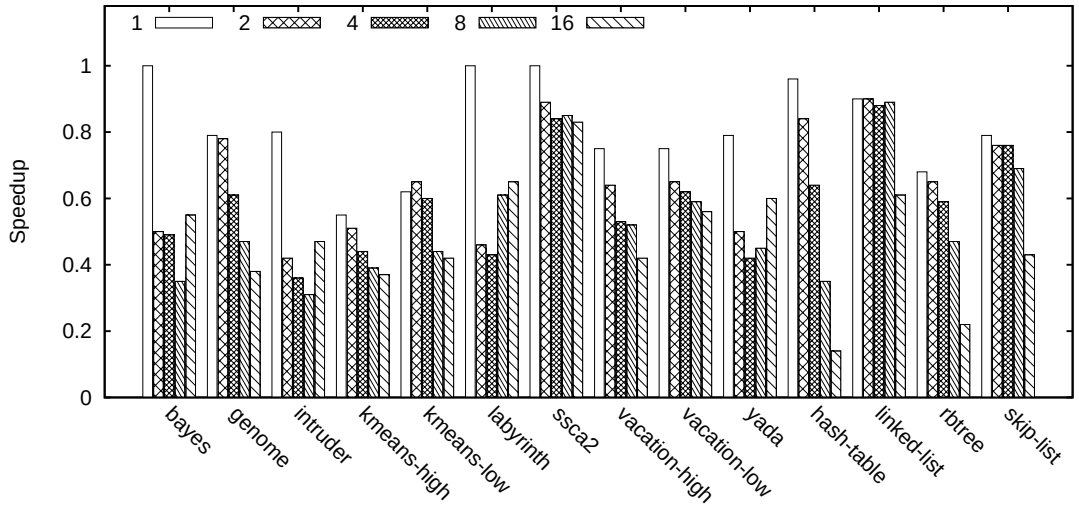


Figure 5.12: Combined overheads of compiler over-instrumentation and transparent privatization on *x86*.

Threads	Min	Max	Avg
1	0	0.45	**0.16**
2	0.1	0.58	**0.35**
4	0.12	0.64	**0.41**
8	0.11	0.69	**0.47**
16	0.17	0.86	**0.52**

Table 5.5: Summary of combined over-instrumentation and transparent privatization overheads on *x86*.

on 8 workloads. Furthermore, the best achieved speedup over sequential code is only 4.5x, compared to SwissTM-ME where it is over 9x.

The overheads of SwissTM-CT compared to SwissTM-ME are presented in detail in Figure 5.12 and Table 5.5. The overheads with 16 threads are sometimes as high as 80%, for example on *hash-table* and *rbtree*, but are also sometimes as low as 20%, for example on *ssca2*. The overheads of SwissTM-CT are largely a simple combination of SwissTM-CE and SwissTM-MT overheads, meaning that both its performance and scalability are impaired. The techniques for reducing transparent privatization and over-instrumentation overheads are applicable to SwissTM-CT as well.

5.8 Programming model

The extensive experiments presented in this section imply that STM-CE variant, where the programmers use compiler for instrumenting the programs, but explicitly require privatization where needed, is the most appropriate programming model for STM in many cases: it performs

and scales well, and exposes a relatively simple programming model, hitting the “sweet-spot” for most programers.

SwissTM-ME performs better than SwissTM-CE, but is probably too tedious and error prone to use in most applications, and might be appropriate only for smaller applications or performance critical sections of code. Without the compiler, it is very likely that there would be too many instances of incorrect manual instrumentation, which would result in bugs that are hard to detect, defeating the main purpose of TM paradigm. For that reason, an STM compiler is crucial for usability.

On the other hand, transparently supporting privatization is not absolutely necessary: privatizing a piece of data is, most often, a conscious decision made by a programmer rather than an accidental occurrence. Furthermore, unlike manual instrumentation, which requires annotations of each memory access, explicit privatization requires only transaction-level annotations, which means that it is not nearly as hard to use correctly as manual instrumentation. This implies that explicitly marking privatizing transactions does not require too great an additional effort from the programmer, and is, therefore, suitable for wider-spread use. The only case when transparent privatization might have certain advantages is when lock-based code is rewritten to use transactions: such code might not work correctly if it uses the privatization idiom. In such cases, additional care is needed by programmers who opt to use an STM-CE system.

5.9 Summary

In this chapter, I reported on the most exhaustive evaluation to date of the STM ability to outperform sequential code. The evaluation uses SwissTM and compares it to sequential code on 17 workloads with widely varying characteristics and two different parallel systems, demonstrating that STM can perform well in a wide range of workloads. Whereas I do not argue that STM is a silver bullet for general purpose concurrent programming, the presented results contradict a recent study, which shows particularly bad performance of STM [19], and suggest that STM is already now usable in practice for various types of programs. The presented results support the initial hopes about STM performance and motivate further research in the field.

Several techniques could further improve the STM performance, making it even more appealing. For example, static segregation of memory locations, depending on whether they are shared or not, can minimize compiler instrumentation overhead, partially visible reads can improve privatization performance, whereas reducing the rate of accesses to shared meta-data can enhance scalability.

6 Related Work

In this chapter, I present the related work. First, I discuss some of my work done in parallel with this thesis, that mostly focuses on the performance of STM. Then, I present related work on the design and implementation of other techniques for improving STM performance.

6.1 My work

Compiler optimizations. As shown in the previous chapter, the overheads of compiler over-instrumentation can be significant. In [38], transaction-local memory is identified as a major source of compiler over-instrumentation overheads in STM. Transaction-local memory is memory allocated inside a transaction, which cannot escape, or is captured by, the allocating transaction. Accesses to such memory do not require calls to STM read and write functions. A compiler unaware of that, however, may translate simple memory read and write instructions accessing such memory into more expensive STM calls. This presents an opportunity to improve performance of code instrumented with an STM compiler. The measurements with the STAMP benchmark suite revealed that as many as 60% of the STM calls generated by a base-line compiler are accesses to captured memory, which include 90% of the writes and 45% of the reads. I proposed runtime and compiler optimizations to elide STM barriers to captured memory. Similar techniques can also be used to elide barriers for accesses to thread-local and read-only data. Those optimizations were implemented in the context of Intel C++ STM compiler [85, 99]. The experiments with the STAMP benchmark suite on a system, with 24 cores in four chips showed that up to 18% performance improvement could be achieved when running with 16 threads. The optimizations were integrated in the compiler I used in the previous chapter.

Specializing STM interface. Recently, I started exploring a new approach to making STM practical by trading some of STM generality for performance [36]. I developed SpecTM, an STM that supports efficient short transactions. To do so, it restricts the interface traditionally exposed by STM and it requires the developers to provide more details about transactional

accesses than in "traditional" STMs used in the previous chapters, such as an index of each access inside a transaction. SpecTM also collocates user data and STM meta-data by using transactional variables. I was able to build efficient and scalable hash-table and skip-list data structures using SpecTM, which perform as well as their lock-free counterparts. They outperform the same data structures built using a well optimized, "traditional" STM by more than 60%, and have performance similar to sequential code with a single thread.

SpecTM is more difficult to use than "traditional" STMs, and is probably not a good fit for the average programmers. However, it can significantly help the expert programmers as it enables them to atomically access a handful of memory locations in their parallel algorithms. This ability makes it much easier to develop efficient data structures than when having to rely only on single-location atomic primitives, such as *compare-and-swap* and *test-and-set*, provided directly by the hardware.

Using HTM. Whereas the performance of STM is good across a range of workloads, it is not a silver bullet for all parallel workloads, as the evaluation in the previous chapter shows. Hence, techniques for improving the performance of STM are needed. One possible approach to improving performance of transactional code is to use the best-effort HTMs, such as Sun's Rock [27], AMD's ASF [4], or Intel's TSX [91], once they become available. It has already been shown that best-effort HTMs can improve the performance of parallel data structures [26, 27]. Furthermore, HTMs support stronger semantics when mixing transactional and non-transactional accesses to data, by supporting strong atomicity, which solves many semantical issues of STM. I showed that having a best-effort HTM can simplify dynamic memory management in concurrent algorithms as well [37]. Interestingly, by using RockTM [27], I was able to produce code that is simpler to write, despite the use of fine-grained transactions, that performs better, and reclaims memory sooner than the state-of-the-art lock-free code.

Predicting scalability of STM. Conducting a thorough performance evaluation of an STM is very time consuming. Depressingly, even with all this effort, and even with the same application, it can still be hard to predict the performance if the number of underlying threads on which the application needs to be deployed is different than those of the experiments. Basically, one might have to conduct an entire set of new experiments to get some understanding of the performance of the STM with the new number of threads.

I proposed a pragmatic approach to contribute to changing this state of affairs in [33]. Using classical engineering approximation techniques, I extract, from a set of STM performance measurements, analytical performance functions to model the scalability of STMs. I showed, more specifically, that polynomial and rational functions provide good interpolations of STM performance: even with only a handful of measurements, the average prediction error in most cases is around 1-2%. Furthermore, I showed that we can perform reasonably precise extrapolations using rational functions: basically, using measurements with up to m threads,

we can predict the performance up to roughly $2m$ threads with a relatively low error of around 10% in best cases.

I also discussed two possible applications of this pragmatic approach: (1) how to statically decide whether to use an STM for a given workload and a given number of threads, and (2) how to dynamically adjust the number of threads that execute in parallel to match the optimal concurrency level of a given workload.

I also extended and applied the same approach to predicting the scalability of general parallel applications, regardless of the synchronization technique they use, by developing *PreSca*, a pragmatic system for predicting the scalability of parallel applications [34].

6.2 Others

Next, I describe several of the most prominent STM designs from the literature, followed by a discussion of STM benchmarks and parallel applications implemented with TM. Then, I describe several techniques for optimizing compilers and improving privatization support. I also discuss approaches that relax transactional semantics to improve STM performance, and several other techniques, including transactional scheduling and parallel nesting of transactions.

6.2.1 STM design

There has been a lot of work on various approaches to building STMs, resulting in a plethora of STM designs. I split the STMs discussed in this section into several loose categories: non-blocking, which include initial dynamic STM designs, multi-versioned STMs, that keep several versions of each object, lock-based STMs, which use locking for writes and typically rely on invisible reads, and value-based STMs that use only global meta-data.

Static transactions. The initial STM proposal [101] exposes a static interface, which is meant to be used for expressing low-level atomic primitives that operate on several memory locations, such as a multi-word compare-and-swap (CASN). This is similar in spirit to direct implementations of CASN primitives, such as [58, 82], but more flexible, as it enables programmers to express different atomic operations if needed. The implementations of the static STM and CASN primitives are typically lock-free, meaning that progress of at least one thread in the system is ensured.

Non-blocking STMs. The first dynamic STM proposal is DSTM [63]. It exposes an object-based interface, similar to the one introduced in Chapter 2. DSTM uses invisible reads and eagerly acquires objects for writing. Instead of using locking, it guarantees obstruction-freedom, meaning that when a thread executes alone it makes progress [61]. DSTM uses

incremental read-set validation on every read to guarantee that transactions always operate on consistent snapshots of memory. The notion of contention manager which resolves conflicts among transactions was also introduced in the context of DSTM. Several contention managers were later proposed and evaluated with DSTM, including Polka [100].

WSTM is a dynamic, word-based STM used as a basis for integrating transactional support in Java [54]. It provides lock-free progress guarantees. Transactions use invisible reads, acquire locations at commit-time and use buffered updates. Meta-data is stored in a separate table of ownership records, where memory locations are mapped to the corresponding ownership records using a hash-based approach. FSTM [46] has a similar design to WSTM, with the exception that it is object-based: transactions use invisible readers, they buffer updates, and acquire objects for writing at commit time. Like WSTM, FSTM is also lock-free. It is written in C/C++ and uses epoch-based memory management to ensure safe memory reclamation in the unmanaged environment.

ASTM [76] is another obstruction-free, object-based STM implemented in Java. It does not use a single conflict detection scheme and instead adapts between different schemes to match the characteristics of the executed workload. The base version of RSTM [77], used in the previous chapters, is also obstruction-free and object-based. It can be configured to use a number of conflict detection and contention management algorithms to best fit the workload at hand. The newer versions of the library also feature lock-based and word-based designs.

Whereas most newer STM designs use locks to reduce the implementation overheads, two obstruction-free STMs were proposed more recently. NZSTM [115] is an object-based obstruction-free STM, that directly updates objects, uses eager acquire for writes, and visible readers. In the common case it stores transactional meta-data in-place, together with the objects, and resorts to a level of indirection only when a transaction is aborted. STM described in [75] uses a similar approach to decouple the fast contention-free path and the slower complex path that uses indirection. It also employs techniques that enable high performance typical of lock-based STMs in its nonblocking design. For example, it uses timestamp-based validation, can be configured to use either redo- or undo-logging, and several conflict detection strategies. These techniques improve the performance of the obstruction-free STM significantly, making it competitive in performance with lock-based STMs, such as TL2.

Multi-versioned STMs. JVSTM [16] proposes the use of multiple object versions to improve the performance of long read-only transactions, which never need to abort with multi-versioning. JVSTM is implemented in Java and it proposes a concept of transactional boxes to support multi-versioning. JVSTM uses lazy detection of write-write conflicts. Interestingly, JVSTM is one of the rare STMs that is actually used in production: it was used to replace locking in a web-based applications. The authors report significant improvements in performance perceived by the users, compared to locking.

LSA-STM [92] is a multi-versioned STM that uses a shared time-base timestamp to ensure read-set consistency, as was concurrently proposed by TL2. It is a lock-free, multi-versioned STM, that can be configured to ensure either snapshot isolation, or opacity. Readers are invisible and the shared timestamp is used to ensure that transactions are always accessing consistent snapshots of memory. Validations are used to extend the validity of the read-sets.

Lock-based STMs. After the initial STM proposals that avoid the use of locks, and typically provide non-blocking progress guarantees, lock-based designs were popularized by [41]. It was argued that lock-based designs offer significant performance advantages over obstruction-free ones, with limitations that are going to disappear in future architectures, or are justified trade-offs, considering significant performance improvements they enable compared to non-blocking designs. To demonstrate the performance benefits of the lock-based STM design, an object-based STM that uses eager acquire and invisible reads was shown to outperform DSTM and FSTM by a significant margin on the common red-black tree and skip-list benchmarks [41].

McRT-STM [99] is another lock-based STM. It exposes a word-based interface, and supports configurable conflict detection granularity that can be either object-based or cache-line-based. It is the first STM to use direct updates and undo-logging to reduce perceived high overheads of deferred updates. McRTM-STM uses invisible reads and eager acquire conflict detection. The evaluation of McRT-STM shows good scalability and performance: it outperforms coarse-grained locking on several micro-benchmarks. Interestingly, it also shows that if locking is replaced with STM in the Sendmail application, where around 10% of the time is spent in critical sections, the performance remains roughly the same. McRT-STM also comes with a transaction-aware memory manager that uses an epoch-based scheme to reclaim memory [66].

A lock-based and word-based STM, called TL, is proposed in [29]. TL supports both eager and lazy acquire conflict detection and it always uses invisible reads. Interestingly, neither in TL nor in McRT-STM transactions validate their read-sets when reading, thus allowing for inconsistencies in transaction execution, and not guaranteeing opacity. TL uses non-faulting load instruction, available on SPARC architecture, to avoid dereferencing invalid pointers. It uses Bloom filters [15] to speed-up write-set lookups, when dealing with read-after-write accesses. Three different mappings of memory locations to locks are proposed: per-object, per-stripe, and per-word, with the conclusion that per-object and per-stripe locking perform the best. The experiments suggest that lazy acquire performs better than eager acquire and that STM often scales better than other fine-grained synchronization approaches. TL also introduces quiescence-based memory management for safe transactional memory deallocation, which essentially first privatizes deallocated data using a privatization barrier, similar to the one used by SwissTM, and then deallocates it.

Bartok-STM [57] uses direct memory updates, similarly to McRT-STM, to eliminate write-log lookups during reads, thus speeding them up significantly. It uses invisible reads and eager

locking. Locks are mapped to data at object-granularity, but logging is performed at the field-granularity. Bartok-STM is implemented in the context of the Bartok runtime, and is closely integrated with the compiler and the runtime, which enables significant optimizations. The experiments show that the single-treaded performance with short transactions is around 50% of the performance of sequential, non-thread-safe code.

TL2 [28] extends the TL algorithm with a global shared timestamp used to ensure consistency of transactions' read-sets without relying on expensive read-set validations, the same as LSA-STM. It uses lazy acquisition for writing, and invisible readers. TL2 supports per-stripe and per-object locking granularities. Similarly to TL, it uses quiescence-based memory management.

TinySTM [43] implements very lightweight reads by combining the use of the shared timestamp to speed up read-set validations and direct updates to eliminate the write-set lookups during reads. Transactions marked as read-only by the programmer are further optimized, as they do not maintain read-sets. TinySTM also proposes the use of hierarchical locking to reduce the costs of read-set validations even further. It uses lock-to-address mapping similar to TL2's per-stripe locking scheme. TinySTM outperforms TL2 in all experiments presented in [43], which were based on the red-black tree and linked-list micro-benchmarks.

Each of the time-based STMs, including TL2, LSA-STM, TinySTM, and SwissTM, can suffer from high contention on the shared timestamp, as all update transaction update the timestamp and all read-only transactions read it. To solve the problem of high contention on the shared time-base, two techniques have been proposed. One is to use a single or multiple synchronized physical clocks [93]. The other is to avoid some of the timestamp updates that are not necessary [69]. For example, transactions that fail to increment the clock using a *compare-and-swap* can safely use the old value of the counter as their commit timestamp.

The STM described in [105] is a word-based STM that uses lazy acquire and invisible reads with the shared timestamp to improve the performance of read-set validations. In contrast to most other lock-based STMs, it uses Polka contention manager to resolve conflicts among transactions. Transactions use hash-tables to index their write-sets and, hence, speed-up the write-set lookups when reading. The read-set extension is also used to extend the read-set validity, similar to TinySTM and LSA-STM. The STM also uses visible read bits in the ownership records to support irrevocable transactions, which are transactions that are guaranteed to commit. The bit enables irrevocable transactions to use visible readers and, thus, win even read-write conflicts. Transaction irrevocability is used to support transaction priorities, and provide rather strong progress guarantees to transactions.

TLRW [30] is a recent STM that takes advantage of large-scale single-chip systems, such as the ones based on Sun's UltraSPARC T2 CPU, by using read-write locking. It uses efficient read-write locks, which assign a byte-sized slot to each potential reader and deal with the overflown readers using a counter. The locks can be of different sizes, to accommodate for different number of readers, but are typically one or two cache-lines in size. Because it uses visible reads, TLRW is privatization safe. As all the threads share the same L2-level cache in

UltraSPARC T2, the increase in cache misses which results from the use of visible reads, does not significantly impact performance. With its fast and simple read and write calls, TLRW outperforms TL2 on several micro-benchmarks on a system using a single CPU. However, the performance is significantly impacted on a system with two CPUs, where TLRW does not scale well.

SkyTM [69] uses semi-visible reads to solve a different problem than TLRW: it increases overheads of STM reads and writes, thus sacrificing some of the performance with lower thread counts, to achieve better scalability at high levels of concurrency. SkyTM implements semi-visible reads with scalable SNZI counters [40], enabling writer transactions to detect conflicts and signal the readers that read-set validations are necessary. SkyTM is inherently privatization-safe, due to its use of semi-visible reads, which removes one of the scalability bottlenecks of traditional invisible-read STMs. The evaluation of SkyTM, based on a hash-table micro-benchmark, shows that TL2 with scalable shared timestamp management scales better and outperforms SkyTM even on a system with four UltraSPARC T2 CPUs, which supports 256 hardware threads. When transparent privatization safety is required, however, SkyTM outperforms TL2, which fails to scale to more than one CPU.

Value-based validation. Unlike the STMs described so far, some STMs use only global meta-data to detect conflicts among transactions. Such STMs typically use value-based validation. For example, RingSTM [109] uses a single ring as a shared meta-data, in which committing transactions store Bloom filters that represent their write-sets. These write-set representations are used to detect conflicts between committing transactions and transactions that concurrently read data being updated. RingSTM is livelock-free and privatization safe by design. The evaluation shows that it outperforms TL2 on some high-contention micro-benchmarks.

JudoSTM [86] is implemented in the context of a binary rewriting framework, called Judo, enabling it to fully sandbox the executed code. To improve performance, JudoSTM does not validate read-sets on each read and, instead, relies on sandboxing to isolate faults. It uses invisible reads and deferred writes. It also buffers reads to avoid races which can sometimes occur when a transaction reads the same location multiple times. To commit, each transaction acquires a versioned lock, performs value-based validation of the read-set and commits its write-log to memory. A similar, but finer-grained technique is also proposed. Read-only transactions avoid acquiring the lock, by checking that no transaction committed before and after commit-time validation. The evaluation based on micro-benchmarks and a small-scale system with four cores shows that the overheads of JudoSTM are significant, compared to coarse-grained locking, but that it outperforms RSTM on several workloads.

NOrec [24] uses a similar approach to JudoSTM. It relies on a single versioned lock and performs value-based validation of read-sets. Unlike JudoSTM, it does not sandbox the executed code, and, instead, transactions validate read-sets on every read to guarantee opacity. The version, contained in the lock, is used to avoid validations when they are not needed, similarly

to the commit counter heuristics. NOrec provides strong semantic guarantees, including livelock-freedom and privatization safety. The performance evaluation on UltraSPARC T2 system shows that NOrec outperforms lazy acquire STM described in [105] on micro-benchmarks, but that it is slower on more realistic STAMP workloads.

Database transactions. It is important to note that many of the techniques used by STM were pioneered by database research [48, 118]. However, the characteristics of database and STM workloads differ significantly, resulting in different designs of the databases and STMs.

Summary. My survey of the STMs from the literature illustrates that there are many possible designs for STM, targeting various workload types and system configurations. These differ from SwissTM as they mostly disregard large transactions, support for which is the main focus of SwissTM. Furthermore, SwissTM is designed to provide good performance across a wide range of workloads, not targeting just one point in the spectrum of workload characteristics. Consequently, none of the presented STMs uses the same combination of techniques as SwissTM: mixed eager-lazy conflict detection and the Two-phase contention manager.

6.2.2 Benchmarks

Several realistic benchmarks other than the ones I used in the presented experiments were proposed. In the following, I overview several of these benchmark and parallel applications implemented with TMs.

SPLASH-2 [119] is a suite of highly parallel applications that was designed for comparing different architectural aspects of shared-memory multi-processor computer systems. In particular, it has been used as a benchmark for a number of HTM systems [20, 21, 83]. The main disadvantage of using SPLASH-2 for evaluating TM is the structure of SPLASH-2 programs: in these programs, threads use fine-grained synchronization during short periods of time and spend most of the time performing demanding calculations on thread-local data. While these access patterns result in high performance, they also result in very simple transactions that mostly access basic data types.

QuakeTM [47] and Atomic Quake [123] implement a server for the multiplayer Quake game using Intel's prototype STM compiler and library [85]. They use both coarse- and fine-grained transaction granularities. The used transactions perform various operations on the large shared data structure and use nesting, calls to standard libraries, and I/O operations. Therefore, these benchmarks represent a good test for STM compilers and runtimes, but they also require STM compiler support. The results presented in the papers demonstrate higher overheads of using STM than the ones presented in Chapter 5.

WormBench [122] is a synthetic STM benchmark written in C# that simulates the Snake game. Each thread drives a different snake in the shared game world. The environment, the

characteristics of the snakes, and the ratio of executed operations can be configured to change the workload characteristics. WormBench does not expose any new realistic workloads itself, but is, instead, meant to be used to simulate the characteristics of other benchmarks. The paper shows a configuration that approximates the characteristics of the *genome* benchmark from STAMP as an example. The evaluation also shows that transactional WormBench using Bartok-STM [57] performs worse than the sequential code even when using 16 threads.

Similarly to Atomic Quake and QuakeTM, SynQuake [73] is based on the Quake game. It is a benchmark that models the core data structures and operations of the Quake server and includes a synthetic workload generator to automatically execute player actions. SynQuake is implemented both using state-of-the-art locking techniques and an STM system called libTM [72]. It is integrated with libTM rather tightly, and is, therefore, not easy to port to other STMs. The evaluation shows that STM scales better than state-of-the-art locking implementation and outperforms it with eight threads by more than 30% on average. This is mostly due to the finer-grained synchronization STM uses, which results in less false sharing.

RMS-TM [68] is a set of applications from recognition, mining, and synthesis (RMS) domain, which are argued to represent the future TM workloads. The applications are implemented both using locks and TM, enabling the comparison of locking and TM performance. Similarly to QuakeTM and Atomic Quake, the transactions execute I/O operations, use nesting, and invoke library calls, therefore extensively testing the TM compiler and system support for executing transactions of various characteristics. RMS-TM was implemented using the Intel's prototype STM compiler and library [85] and two HTMs [20, 116]. The evaluation demonstrates that the benchmarks scale well up to eight threads, exhibiting better scalability than the STAMP benchmarks with all three used TM systems.

6.2.3 Compiler optimizations

As was shown in [121] and in the previous chapter, using an STM compiler introduces certain over-instrumentation overheads. Not surprisingly, several compiler optimizations that target these and other STM overheads were proposed.

Bartok-STM [57] proposes a number of extensions to the STM compiler to reduce various overheads. For example, it decomposes library's transactional interface and extends the existing compiler optimizations to take advantage of the decomposed interface when possible. Also, it relies on inter-procedural analysis to eliminate STM calls to objects allocated inside the current transaction. The analysis is complemented with runtime log filtering. The compiler also tries to avoid read to write upgrades, that occur when object is accessed first for reading and then for writing. In those cases, it opens the object for writing immediately at read time, eliminating all costs of the read. Other optimizations include moving the common calls for opening objects from invoked function to the caller and further decomposing the log management functions. The presented evaluation confirms that, although individual

optimizations rarely result in great improvements by themselves, the effects add up, resulting in noticeably better performance when they are combined.

The Java STM compiler described in [3] uses similar STM interface decomposition as Bartok-STM to leverage the existing compiler optimizations. It also eliminates STM calls for reads of constant data, such as fields declared as `final` in Java, and STM calls for reads and writes to data allocated inside the current transaction. It optimizes read-after-write access patterns to the same object by proactively acquiring the object for writing at the time of the read, similarly to Bartok-STM. Furthermore, its just-in-time compiler inlines fast-paths of STM calls in the generated native code to avoid costs of function calls. Similarly to the results presented for Bartok-STM, the evaluation shows that the proposed optimizations significantly reduce the overheads with a single thread, producing the code that is sometimes only 16% slower with short transactions than coarse-grained locking.

The Intel's C/C++ STM compiler described in [85], uses optimizations that aim to avoid unnecessary read to write upgrades, similarly to what Bartok-STM compiler and Java compiler from [3] do. It extends this approach to also optimize write-after-write and read-after-read accesses to the same locations. It, furthermore, supports explicit annotations of functions and code blocks that do not require STM instrumentation to improve their performance. I used such annotations to optimize STAMP, where indicated by the original STAMP implementation.

The compiler described in [94] partitions program data based on the runtime program characteristics to enable fine-tuning of STM algorithm for different partitions. For example, it uses no concurrency control for read-only data partitions, a single lock for partitions that are rarely updated, and the general-purpose STM algorithm for partitions that have more general read-write access patterns. The presented evaluation, based on a subset of STAMP, shows that this approach can sometimes indeed improve performance of the resulting programs.

The compiler from [39] proposes several compiler optimizations in the context of an object-based STM. It, for example, uses data-flow analysis to detect many cases of read-after-write and write-after-write access patterns in the same transaction, even if they are not close to each other in the source code. This work proposes optimizations both for the locking STMs, as typically proposed by others, but also for obstruction-free STMs.

The compiler used in the previous chapter uses some of the optimizations that were mentioned. In particular, it uses all optimizations from [85], as well as some optimizations proposed in [38].

6.2.4 Privatization

The evaluation from the previous chapter shows that transparently supporting privatization for STMs that use invisible readers and per-location meta-data incurs high overheads and impairs scalability. To reduce these overheads, the use of partially visible readers is proposed in [78]. This approach relies on the shared time-base, used for optimizing the read-set validations. On every read, transactions store their current timestamp into the *orec* of the location they

are reading. The store can be omitted if the meta-data already contains a higher timestamp, reducing the cost of the reads compared to fully visible readers. The writers need to execute the privatization barrier after they commit only when one of the locations they updated is actually being accessed by a concurrent reader, which is indicated by the timestamp stored in its *orec*. This reduces the cost of the transparent privatization. The evaluation shows relatively high overheads: up to 50% with 32 threads when compared to TL2 that is not privatization safe.

It is worth repeating that some STM designs are, inherently, privatization safe. STMs that use visible reads, such as TLRW [30], or semi-visible reads, such as SkyTM [69], do not suffer from this problem. Read calls of these STMs, however, are typically expensive, as they update shared meta-data. Also, STMs that rely only on centralized meta-data, such as RingSTM [109], JudoSTM [86], and NOrec [24], are typically privatization safe. These STMs, however, are best suited to workloads consisting of short transactions, as their scalability can be impacted by transactions with long commits.

6.2.5 Relaxed transactions

Several relaxed models have also been proposed to improve STM performance. These are typically aimed at more skilled programmers, as they are more difficult to use than traditional STMs.

Early release [46, 63] allows programmers to manually remove objects from transactions' read- and write-sets. Removing the objects from read-sets reduces validation costs and can also help avoid unnecessary conflicts in some cases. It is particularly appealing when implementing data structures such as linked-list, where the updates to a node only depend on several neighboring nodes. In these data structures, a traditional transaction contains in its read-set all the nodes preceding the node being updated. With early release, the validation costs and conflict probability are reduced, which improves performance overall. Using the early release, however, can be difficult as programmers can easily write incorrect code if they are not careful, and is thus better suited to experts than average programmers.

Open nesting [84] supports transaction nesting where inner transactions can commit independently from the enclosing transaction. Open nesting introduces abstract locks to detect conflicts at the semantical level, instead of the level of memory accesses. For example, two transactions that, during their execution, insert different elements in a set do not conflict at the semantical level, but they might conflict at the level of memory locations they access. This happens when, for example, the two elements map to the same bucket in the hash-table set implementation. With open nesting, inner transactions are used to insert the elements into the set and they are allowed to commit, whereas the enclosing transactions only access the abstract locks corresponding to the inserted elements. Consequently, the two top-level transactions do not conflict as long as they access different abstract locks. This can significantly improve the scalability of programs by allowing more inter-leavings between threads. The

evaluation presented in [84], based on the set micro-benchmarks, shows that open nesting can indeed improve scalability compared to closed and flat nesting.

Transactional boosting [59] has a similar goal to open nesting: it uses lock-based implementations of common data structures inside transactions to improve performance of transactions that access these objects. Similarly to open nesting, transactional boosting relies on abstract locks to detect conflicts at the semantical level between transactions that access the same lock-based objects. To rollback the updates of these objects when aborting, transactions log the performed operations and execute compensating actions on aborts. A significant advantage of transactional boosting over open nesting is that the operations on the low-level data structures are typically more efficient as they use locking instead of transactions. The presented evaluation shows that boosting can indeed improve STM performance: on a system with eight threads it speeds-up TL2 on *vacation* from STAMP by around 2x, and on *kmeans* by between 1.5x and 2x.

Elastic transactions [44] are similar to early release in that they only require a subset of accesses performed by transaction to be serialized. Under certain circumstances, elastic transactions can simply discard part of their read-set upon conflict and continue with the execution. Elastic transactions are well suited to linear data structures, such as linked-lists, hash-tables, and skip-lists, but they were also successfully applied to red-black trees. In general, elastic transactions are rather difficult to apply to general data structures, and are, thus, better suited to concurrency experts. The evaluation shows that elastic transactions improve performance of STM by 36% on average on the micro-benchmarks.

I do not use any form of relaxed transactions with SwissTM, as I focus on performance of more traditional STMs. Techniques such as transactional boosting and early release would be straightforward to implement in the context of SwissTM, and they might help improve performance on some workloads.

6.2.6 Other techniques

Several STMs propose switching between different algorithms at runtime to adapt to workload characteristics. As mentioned, ASTM [76] adapts to workload characteristics, by maintaining history of access patterns of previous transactions and switching between eager and lazy object acquire. The STM described in [85] supports switching between invisible readers, visible readers, and use of a single lock on per-transaction basis. To do so, it employs a level of indirection when invoking the STM calls, enabling switching between various schemes at runtime. The contention manager keeps switching to progressively less optimistic techniques after a transaction repeatedly aborts to increase its chance of success. A similar, lightweight adaptivity approach based on tables of function pointers, is described in [104]. It enables several interesting optimizations, where transactions can start as read-only, which eliminates read-after-write checks for example, and can transition into read-write transactions at the time of the first write by changing the function pointer table. The paper describes another

interesting usage of adaptivity, where transaction switch to progressively less optimistic modes upon aborts, but using many more STM variants than in [85]. Despite several proposals, adaptively switching between different STM variants based on the workload characteristics has yet to prove it can improve performance of STM by any significant margin. Hence, it is not used in SwissTM for anything more than for adaptively switching between the two contention management phases.

DASTM [90] extends TL2 algorithm to implement STM that ensures conflict serializability [48] instead of serializability. With DASTM, when a transaction detects a conflict, it, instead of aborting, acquires a dependency on the conflicting transactions. The dependencies restrict the commit order of transactions, and in cases of read-write conflicts, require forwarding of uncommitted data between transactions. Experimental results show that DASTM can outperform TL2 by up to 4.8x on high contention workloads, such as a shared counter benchmark and several high contention STAMP workloads.

Several papers propose transaction scheduling to proactively react to possible conflicts between transactions, based on the history of past conflicts. The main difference between the scheduler and the contention manager is that the former decides whether to execute a transaction or wait before the transaction starts, and the latter decides how to resolve the conflict when it already occurs. A scheduler called Shrink is presented in [35].[1] It uses the access patterns of the previous transactions from the same thread to predict the locations which the future transaction will access, and postpones transaction execution if there is too much contention on that data. It improves performance, but typically only when the system is overloaded. Similarly, a scheduler described in [120], detects when contention is too high and serializes threads by storing them in a queue. In steal-on-abort approach for transactional scheduling [9], each aborted transaction gets enqueued behind the transaction that caused the abort, thus eliminating repeated aborts due to the same conflict. CAR-STM [31] proposes a similar approach, but it extends it to enable users to assign per-transaction collision probabilities used to schedule the transaction prior to abort. The described scheduling approaches can help improve the performance of SwissTM in certain cases, and, in fact, Shrink was integrated with SwissTM, but transactional scheduling is largely orthogonal to the main focus of this thesis.

A transaction-aware, kernel-level scheduler, proposed in [74], enables lightweight communication between the kernel scheduler and STM through a common memory region, with the goal of reducing repeated aborts. For example, it enables limited time-slice extensions for threads that are currently executing a transaction, to prevent internal STM locks from being held by a transaction that is blocked by the operating system.

Parallel nesting of transactions, where nested transactions execute concurrently, has also been proposed as a way to better utilize increasing numbers of threads supported by modern processors. NePaLTM [117] integrates nested parallel transactions with OpenMP, but allows

[1] I was involved in development of Shrink.

only limited interaction between nested transactions. For example, it serializes sibling transactions that might access the same data. A parallel-nested STM prototype described in [12][2] implements efficient ancestor queries in a manner similar to what was proposed in [7]. It is argued that efficient ancestor queries are important to support deep nesting hierarchies that are likely to occur in complex programs. An evaluation of the prototype also shows that parallel nesting makes sense in certain scenarios, but does not evaluate overheads of implementing it in detail. NesTM [11] is another STM that supports parallel nested transactions. It demonstrates that the overheads of supporting parallel nesting can be reasonably low, and that program performance can be improved by using parallel nesting in certain scenarios. Whereas parallel nesting seems promising, the current techniques have only limited positive impact on performance.

[2] I was involved in development of this parallel nesting STM.

7 Conclusions

In this thesis, I described algorithmic and implementation techniques for software transactional memory that efficiently support workloads consisting of large transactions. Such workloads are of great importance if STM is to support large-scale applications, such as video games, application and web servers. Surprisingly, support for large transactions has mostly been neglected by previous work, as demonstrated in Chapter 3.

In Chapter 4, I proposed SwissTM, an effective mix of design choices and implementation techniques, that achieves good performance on workloads with large transactions, while, at the same time, not compromising on the performance with smaller transactions. SwissTM uses the *mixed* eager-lazy conflict detection to detect write-write conflicts among transactions eagerly and read-write conflicts lazily, and the *Two-phase* contention manager to incur low overheads on short and read-only transactions, while still providing effective contention manager for longer ones. Whereas I do not claim that SwissTM is the best choice for all imaginable workloads, the extensive evaluation I presented shows that the proposed techniques are indeed effective in many cases: (1) SwissTM outperforms state-of-the-art STMs on large-scale workloads of STMBench7 and (2) it either outperforms them, sometimes significantly, or matches their performance on a wide range of workloads with smaller transactions. Furthermore, I "dissected" the design of SwissTM by evaluating performance impact of each of the design choices I made. This analysis enables other STM designers to understand the individual impact of these choices under different conditions, allowing them to reuse the techniques that best fit their own STM designs.

I demonstrated that STM is indeed a viable choice for writing parallel applications today by presenting an extensive comparison of SwissTM performance to performance of sequential code in Chapter 5. The goal of the evaluation is to understand whether STM-based code can outperform sequential code, and if it can, how many CPUs it requires to do so. After all, writing parallel code using STM requires only slightly more effort than writing the equivalent sequential code, and if the performance is better, the programmers might be tempted to use STM even though it does not match the performance of more involved synchronization techniques. The results show that the performance of STM is much better than previously

claimed: in most cases, parallel code based on SwissTM indeed overcomes high overheads of book-keeping in software and outperforms sequential code, often with only a handful of threads.

The evaluation also considers various levels of support for privatization idiom and compiler code instrumentation, thus improving the understanding of their respective costs and making it possible to weigh their costs against the benefits. The results lead me to conclude that existing techniques for ensuring privatization safety are too costly, in particular because they do not scale well. On the other hand, the overheads of over-instrumentation introduced by STM compilers are not as high and, more importantly, they remain roughly the same at different thread counts. Furthermore, the benefits of using an STM compiler seem much higher than the benefits of providing transparent privatization support. Taking all this into account, I conclude that the most promising STM variant uses an STM compiler to instrument programs, but requires programmers to explicitly mark privatizing transactions. Such an STM variant delivers good performance while exposing relatively clean programming abstraction, and, thus, hits a "sweet spot" for most programmers.

In conclusion, in this thesis I proposed SwissTM, a software transactional memory that effectively supports workloads that use large-scale transactions, alongside of smaller-scale ones, and demonstrated that software transactional memory can indeed be used in practice to write parallel code that performs well on a range of workloads.

Future work. Whereas the performance of STM is often good, it is certainly not always impressive. Consequently, it has to be improved further for STM to be more widely adopted. Along with the new techniques that will boost the performance, we will need pragmatic models and tools for predicting the performance of STM-based programs. Such tools are needed to enable programmers predict performance of STM-based programs early in the development cycle thus helping them decide whether to use STM for a particular programming task or not.

In the future, we are very likely to see increased interest in algorithms and data structures that rely on very short transactions, possibly much shorter than transactions in the currently used micro-benchmarks. Interest in such algorithms will be stirred by the upcoming best-effort HTMs, in particular by Intel's TSX [91]. Such HTMs typically support only short transactions, and new concurrent algorithms are needed to fully utilize them. I have already started investigating support for short software transactions: by specializing STM interface, I was able to implement efficient short transactions at the cost of making the interface more awkward to use [36]. The idea of short, specialized transactions in software is certainly worth exploring further: such transactions enable us to develop algorithms in anticipation of HTM, but benefit from them immediately, even before HTM is available. Interesting directions for future work include exploration of new forms of STM specialization and integration of the specialized transactions with compilers. Such integration would improve the performance

of the compiler-generated code, without forcing programmers to use awkward specialized interfaces.

The best-effort nature of forthcoming HTMs will also make hybrid TM systems increasingly more important. Building STMs that perform well on their own as well as when they are integrated with an HTM is a challenging task, as the characteristics of the HTM significantly impact the design of the STM. For example, using a simple single-lock implementation of STM might be justified with hardware that supports transactions of at least moderate sizes. However, with the first generations of HTMs, more sophisticated STM designs will be needed to ensure progress and good performance in face of many transactions that hardware does not support.

Looking at the broader picture, researchers will face many challenges in trying to fully exploit future large-scale systems that support hundreds or even thousands of hardware threads. To support scaling of that magnitude, we will have to further improve our understanding of how to structure parallel applications and which programming constructs to use. Novel algorithms and data structures will be the key in achieving this goal: as the number of hardware threads keeps increasing, new scalability bottlenecks will get exposed in the data structures and algorithms commonly used today, making them inadequate for future systems. In fact, some, if not most, data structures commonly used nowadays are a better fit for sequential than for parallel programs. To fully exploit the future large-scale parallel systems we will have to replace them with fundamentally different data structures instead of simply implementing them using different techniques, such as transactional memory. For instance, queues and stacks are not very scalable as they impose linear ordering between the stored data elements. On the other hand, pools can often be used in their place, and scalable implementations of pools exist [6, 13]. Similarly, a fetch-and-increment counter that is used only to track whether a certain value is equal to zero or not can be replaced by a more scalable SNZI data structure [40]. These two examples represent the sort of radical changes we will see in the design of future, highly-scalable data structures: relaxed, non-deterministic data structures will replace the, today predominant, deterministic data structures that were optimized for execution on obsolete sequential systems. Transactional memory is likely to play an important role in building these new concurrent algorithms, as it greatly simplifies their design and implementation.

Bibliography

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 63–74, 2008.

[2] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Unlocking concurrency. *ACM Queue*, 4:24–33, December 2006.

[3] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 26–37, 2006.

[4] Advanced Micro Devices, Inc. Advanced synchronization facility—proposed architectural specification 2.1, 2009.

[5] Y. Afek, U. Drepper, P. Felber, C. Fetzer, V. Gramoli, M. Hohmuth, E. Riviere, P. Stenstrom, O. Unsal, W. M. Moreira, D. Harmanci, P. Marlier, S. Diestelhorst, M. Pohlack, A. Cristal, I. Hur, A. Dragojević, R. Guerraoui, M. Kapalka, S. Tomic, G. Korland, N. Shavit, M. Nowack, and T. Riegel. The Velox transactional memory stack. *IEEE Micro*, 30:76–87, September 2010.

[6] Y. Afek, M. Hakimi, and A. Morrison. Fast and scalable rendezvousing. In *Proceedings of the 25th International Conference on Distributed computing*, DISC'11, pages 16–31, 2011.

[7] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 163–174, 2008.

[8] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Lujan, and K. Jarvis. Lee-TM: A non-trivial benchmark suite for transactional memory. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '08, pages 196–207, 2008.

[9] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering.

In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 4–18, 2009.

[10] The atomic_ops project. http://www.hpl.hp.com/research/linux/atomic_ops.

[11] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Implementing and evaluating nested parallel transactions in software transactional memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 253–262, 2010.

[12] J. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka. Leveraging parallel nesting in transactional memory. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 91–100, 2010.

[13] D. Basin, R. Fan, I. Keidar, O. Kiselov, and D. Perelman. CAFÉ: scalable task pools with adjustable fairness and contention. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, pages 475–488, 2011.

[14] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, 1995.

[15] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, July 1970.

[16] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63:172–185, December 2006.

[17] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the 2008 IEEE International Symposium on Workload Characterization*, IISWC '08, pages 35–46, 2008.

[18] M. J. Carey, D. J. DeWitt, C. Kant, and J. F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Proceedings of the 9th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '94, pages 414–426, 1994.

[19] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *ACM Queue*, 6:46–58, September 2008.

[20] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 97–108, 2007.

[21] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, pages 347–358, 2006.

[22] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 39–52, 2011.

[23] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott. Capabilities and limitations of library-based software transactional memory in C++. In *the 2nd ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[24] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, 2010.

[25] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, pages 336–346, 2006.

[26] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 325–334, 2010.

[27] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pages 157–168, 2009.

[28] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, DISC '06, pages 194–208, 2006.

[29] D. Dice and N. Shavit. What really makes transactions faster? In *the 1st ACM SIGPLAN Workshop on Transactional Computing*, Transact '06, 2006.

[30] D. Dice and N. Shavit. TLRW: return of the read-write lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 284–293, 2010.

[31] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the 27th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '08, pages 125–134, 2008.

[32] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. Technical Report LPD-REPORT-2009-003, EPFL, 2009.

[33] A. Dragojević and R. Guerraoui. Predicting the scalability of an STM: A pragmatic approach. In *the 5th ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT '10, 2010.

[34] A. Dragojević and R. Guerraoui. A pragmatic approach for predicting scalability of parallel applications, 2012. EPFL Technical report EPFL-REPORT-174869.

[35] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the 28th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '09, pages 7–16, 2009.

[36] A. Dragojević and T. Harris. STM in the small: Trading generality for performance in software transactional memory. In *Proceedings of the 7th ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '12, pages –, 2012.

[37] A. Dragojević, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 99–108, 2011.

[38] A. Dragojević, Y. Ni, and A.-R. Adl-Tabatabai. Optimizing transactions for captured memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 214–222, 2009.

[39] G. Eddon and M. Herlihy. Language support and compiler optimizations for STM and transactional boosting. In *Distributed Computing and Internet Technology*, volume 4882 of *Lecture Notes in Computer Science*, pages 209–224. 2007.

[40] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: scalable nonzero indicators. In *Proceedings of the 26th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '07, pages 13–22, 2007.

[41] R. Ennals. Software Transactional Memory Should Not Be Obstruction-Free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, 2006.

[42] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19:624–633, November 1976.

[43] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 237–246, 2008.

[44] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proceedings of the 23rd international Conference on Distributed computing*, DISC'09, pages 93–107, 2009.

[45] P. Felber, T. Riegel, C. Fetzer, M. Süsskraut, U. Müller, and H. Sturzrehm. Transactifying applications using an open compiler framework. In *the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Transact '07, 2007.

[46] K. Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003.

[47] V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Quaketm: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 126–135, 2009.

[48] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., 1st edition, 1992.

[49] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management. In *Proceedings of the 19th International Symposium on Distributed Computing*, DISC '05, pages 303–323, 2005.

[50] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '05, pages 258–264, 2005.

[51] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, 2008.

[52] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '07, pages 315–324, 2007.

[53] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 102–, 2004.

[54] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 388–402, 2003.

[55] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory.* Morgan & Claypool, 2nd edition, 2010.

[56] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, 2005.

[57] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 14–25, 2006.

[58] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 265–279, 2002.

[59] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, 2008.

[60] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems*, 23:146–196, May 2005.

[61] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–529, 2003.

[62] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 253–262, 2006.

[63] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, 2003.

[64] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, 1993.

[65] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programmming Languages and Systems*, 12:463–492, July 1990.

[66] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 74–83, 2006.

[67] Intel Corporation. Intel transactional memory compiler and runtime application binary interface, 2008. Revision 1.0.1.

[68] G. Kestor, S. Stipic, O. S. Unsal, A. Cristal, and M. Valero. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *the 4th ACM SIGPLAN Workshop on Transactional Computing*, Transact '09, 2009.

[69] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *the 4th ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT '09, 2009.

[70] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *the 2nd ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT '07, 2007.

[71] LPD-EPFL. Transactions@epfl. http://lpd.epfl.ch/transactions/.

[72] D. Lupei, A. Czajkowski, C. Segulja, M. Stumm, and C. Amza. Automatic adaptation of transactional memory state management to application conflict patterns. In *the 13th Workshop on Interaction Between Compilers and Computer Architectures*, Interact '09, 2009.

[73] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. Amza. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *Proceedings of the 5th ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '10, pages 41–54, 2010.

[74] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 79–90, 2010.

[75] V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 227–236, 2008.

[76] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, DISC '05, pages 354–368, 2005.

[77] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *the 1st ACM SIGPLAN Workshop on Transactional Computing*, Transact '06, 2006.

[78] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *Proceedings of the 37th International Conference on Parallel Processing*, ICPP '08, pages 67–74, 2008.

[79] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5:17–20, July 2006.

[80] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15:491–504, June 2004.

[81] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 69–80, 2007.

[82] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '97, pages 219–228, 1997.

[83] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, HPCA '06, pages 254–265, 2006.

[84] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 68–78, 2007.

[85] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems Languages, and Applications*, OOPSLA '08, pages 195–212, 2008.

[86] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 365–375, 2007.

[87] V. Pankratius and A.-R. Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 43–52, 2011.

[88] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26:631–653, October 1979.

[89] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 494–505, 2005.

[90] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 163–172, 2009.

[91] J. Reinders. Transactional synchronization in Haswell, Feb. 2012. http://software.intel.com/en-us/blogs.

[92] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing*, DISC '06, pages 284–298, 2006.

[93] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '07, pages 221–228, 2007.

[94] T. Riegel, C. Fetzer, and P. Felber. Automatic data partitioning in software transactional memories. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 152–159, 2008.

[95] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: the importance of nonspeculative operations. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 53–64, 2011.

[96] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. TxLinux: using and managing hardware transactional memory in an operating system. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 87–102, 2007.

[97] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, 2010.

[98] F. Rubin. The Lee path connection algorithm. *IEEE Transactions on Computers*, 23:907–914, September 1974.

[99] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 187–197, 2006.

[100] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, 2005.

[101] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, 1995.

[102] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 78–88, 2007.

[103] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[104] M. F. Spear. Lightweight, robust adaptivity for software transactional memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 273–283, 2010.

[105] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 141–150, 2009.

[106] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the 26th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '07, pages 338–339, 2007.

[107] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory, 2007. Department of Computer Science, University of Rochester Technical Report 915.

[108] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for softwaretransactional memory. In *Proceedings of the 20th International Symposium on Distributed Computing*, DISC '06, pages 179–193, 2006.

[109] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 275–284, 2008.

[110] Standard Performance Evaluation Corporation. Specjbb2000 benchmark, 2000.

[111] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 335–344, 2011.

[112] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, Dec. 2004.

[113] H. Sutter. Welcome to the jungle. Dec. 2011.

[114] T. Sweeney. The next mainstream programming language: a game developer's perspective. Invited talk at the 31st Annua ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'06, 2006.

[115] F. Tabba, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang. NZTM: nonblocking zero-indirection transactional memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 204–213, 2009.

[116] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 145–155, 2009.

[117] H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. NePaLTM: Design and implementation of nested parallelism for transactional memory systems. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, ECOOP '09, pages 123–147, 2009.

[118] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., 2001.

[119] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, 1995.

[120] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 169–178, 2008.

[121] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 265–274, 2008.

[122] F. Zyulkyarov, A. Cristal, S. Cvijic, E. Ayguade, M. Valero, O. Unsal, and T. Harris. WormBench: a configurable workload for evaluating transactional memory systems. In *Proceedings of the 9th Workshop on MEmory Performance: DEaling with Applications, Systems and Architecture*, MEDEA '08, pages 61–68, 2008.

[123] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 25–34, 2009.

Aleksandar Dragojević
Serbian nationality, born 7th October 1980, married
Phone: +41 78 662 92 17
Email: aleksandar.dragojevic@gmail.com

Education

2012	PhD in Computer Science at EPFL, Switzerland
2006 – 2012	Computer Science studies at EPFL, Switzerland
2004	Graduate Engineer in Electrical and Computer Engineering, University of Novi Sad
1999 – 2004	Computer Science Engineering Studies at FTN, University of Novi Sad

Experience

2011 **Microsoft Cambridge, UK** (3-month internship)
Developed a specialized software transactional memory (STM) that trades some generality of traditional STMs for performance. The hash table and skip list algorithms based on the specialized STM are simpler than the lock-free algorithms, but they perform essentially as well, while outperforming "traditional" STM by 60%.

2010 **Oracle Burlington, USA** (3-month internship)
Worked on debugging and optimization of Oracle C/C++ transactional memory (TM) runtime library and compiler. Also developed hardware TM-based algorithms that outperformed the equivalent non-TM algorithms in all cases; even with the best-effort hardware TM I had at my disposal.

2008 **Intel Santa Clara, USA** (3-month internship)
Worked on runtime and compiler optimizations for Intel C/C++ STM runtime system and compiler. The optimizations I proposed were implemented and deployed as a part of Intel's research C/C++ STM-enabled compiler and their STM library and delivered up to 15% performance improvements.

2005 – 2006 **Levi9 Novi Sad, Serbia**
Senior software developer, technical team leader, and project manager for a team of six developers. Decided on technical questions and created long- and short-term plans for the project. Extensively communicated with the clients to specify the requirements. Kept the project running smoothly despite unclear specifications.

2004 – 2005 **Navigator Novi Sad, Serbia**
Software developer on a complex multi-tier ERP solution (several thousand classes). Often solved technically challenging tasks that required obtaining new knowledge, including: integration of 3rd party OLAP component, transparent replacement of the grid component and custom report generation.

Main Interests

I am interested in many areas of computer science, but mainly in concurrent and distributed computing. I prefer staying on the practical side of problems and enjoy implementing real systems, ranging from simple script based solutions to everyday problems to complex production systems.

Publications and Workshops

- A. Dragojević, T. Harris. *STM in the small: trading generality for performance in software transactional memory*. EuroSys '12 the European Conference on Computer Systems, Bern, April 11-13, 2012.
- A. Dragojević, M. Herlihy, Y. Lev and M. Moir. *On The Power of Hardware Transactional Memory to Simplify Memory Management*. 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC 2011), San Jose, California, USA, June 6-8, 2011.
- A. Dragojević, P. Felber, V. Gramoli and R. Guerraoui. *Why STM can be more than a Research Toy*, Communications of the ACM, vol. 54, p. 70-77, April 2011.
- A. Dragojević and R. Guerraoui. *Predicting the Scalability of an STM: A Pragmatic Approach*. 5th ACM SIGPLAN Workshop on Transactional Computing (Transact 2010), Paris, France, April 13, 2010.
- J. Baretto, A. Dragojević, P. Ferreira, R. Guerraoui and M. Kapalka. *Leveraging Parallel Nesting in Transactional Memory*. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010), Bangalore, India, January 9-14, 2010.
- A. Dragojević, Y. Ni and A.-R. Adl-Tabatabai. *Optimizing Transactions for Captured Memory*. 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA 2009), Calgary, AB, Canada, August 11-13, 2009.
- A. Dragojević, A. Singh, R. Guerraoui and V. Singh. *Preventing versus Curing: Avoiding Conflicts in Transactional Memories*. 28th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2009), Calgary, Alberta, Canada, August 10-12, 2009.
- A. Dragojević, R. Guerraoui and M. Kapalka. *Stretching Transactional Memory*. ACM SIGPLAN 2009 Conference on Programming Languages Design and Implementation (PLDI 2009), Dublin, Ireland, June 15-20, 2009.
- A. Dragojević, R. Guerraoui and M. Kapalka. *Dividing Transactional Memories by Zero*. 3rd ACM SIGPLAN Workshop on Transactional Computing (Transact 2008), Salt Lake City, Utah, USA, February 23, 2008.