Optimisation de la performance des applications de mémoire transactionnelle sur des plates-formes multicoeurs : une approche basée sur l'apprentissage automatique

Márcio Bastos Castro Castro

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Mathématiques-informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Márcio BASTOS CASTRO

Thèse dirigée par **Jean-François MÉHAUT**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Improving the Performance of Transactional Memory Applications on Multicores : A Machine Learning-based Approach

Thèse soutenue publiquement le **3 décembre 2012**,
devant le jury composé de :

M. Philippe O. A. NAVAUX
Professeur, Universidade Federal do Rio Grande do Sul (UFRGS), Président
M. Pascal FELBER
Professeur, Université de Neuchâtel, Rapporteur
M. Raymond NAMYST
Professeur, Université de Bordeaux 1, Rapporteur
M. Miguel SANTANA
Directeur du centre IDTEC, STMicroelectronics, Crolles, Examinateur
M. Jean-François MÉHAUT
Professeur, Université de Grenoble - CEA, Directeur de thèse
M. Luiz Gustavo L. FERNANDES
Professeur, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Co-Encadrant de thèse

Acknowledgements

"When we least expect it, life sets us a challenge to test our courage and willingness to change; at such a moment, there is no point in pretending that nothing has happened or in saying that we are not yet ready. The challenge will not wait. Life does not look back. A week is more than enough time for us to decide whether or not to accept our destiny." — Paulo Coelho, *The Devil and Miss Prym*, 2000.

This extract from a book written by Paulo Coelho (a Brazilian novelist and one of the most widely read authors in the world) summarizes what happened in my life in early 2009. I had the opportunity to meet Jean-François Méhaut in Porto Alegre, Brazil. After a short presentation of the research I'd done during my Master's, Jean-François asked me if I would be interested in doing a Ph.D. in France. Deciding whether to do a Ph.D. in a foreign country is tough. It's a huge investment in terms of time and effort. It puts several technical and personal skills to the test such as the ability to: do good quality research, absorb a new culture, learn a foreign language and live far away from home. I accepted the challenge and it is with great happiness that I write these words to thank many people involved in that.

Firstly, I would like to thank my advisor Jean-François Méhaut, for accepting me as his Ph.D. student and for giving me the opportunity to prepare and write this thesis in France. You were always present to discuss about my research and to motivate me during all this time. I really appreciate your efforts to make collaborations with Brazilian universities. This is very important, since most of us (Brazilians) come back to Brazil after finishing the thesis. Thank you for giving me all the support I needed to participate and present my work in different conferences and also for putting me in contact with many people from the HPC community in France and abroad.

Secondly, I would like to thank all my friends and colleagues from Brazil (PUCRS, UFRGS, PUC Minas and USP) and France (LIG). Special thanks to my big friend Pedro Velho, who helped me a lot when I arrived in France as well as the people from Nanosim, Mescal and Moais research teams for the technical and personal discussions during these years. I really appreciated to share these moments with you!

Finally, I would like to thank my family for supporting me during all my life. A very special thank to my love, Evanaska, who always encouraged me and helped me in all possible ways to achieve this goal. I dedicate this achievement to you all.

Muito obrigado!

Contents

List of Figures

List of Tables

List of Abbreviations

DBMS	Transactional Database Management Systems
DRAM	Dynamic Random Access Memory
GCC	GNU Compiler Collection
GFlops	Billions of floating-point operations per second
HTM	Hardware Transactional Memory
hwloc	Hardware Locality
HyTM	Hybrid Transactional Memory
ID3	Iterative Dichotomiser 3
LLC	Last Level Cache
ML	Machine Learning
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
PAPI	Performance Application Programming Interface
STAMP	Stanford Transactional Applications for Multi-Processing
STM	Software Transactional Memory
TAU	Tuning and Analysis Utilities
TM	Transactional Memory
TSC	Time Stamp Counter
TSP	Traveling Salesman Problem
UMA	Uniform Memory Access

CHAPTER 1

Introduction

THERE WAS A 30-YEAR PERIOD in which the advances in semiconductor technology and computer architectures improved the performance of a single processor at a high annual rate of 40% to 50% [LK08]. However, some issues such as dissipating heat from increasingly densely packed transistors begun to limit the rate at which processor frequencies could be increased. This was one of the reasons why most semiconductor industries are now investing in multicore processors.

Indeed, multicore processors are a mainstream approach to deliver higher performance to parallel applications [Asa+09].Unfortunately, the growing disparity between how fast a processor can operate on data and how fast it can get the data it needs leads to the so-called "memory wall problem" [McK04]. Consequently, these platforms usually feature complex memory hierarchies composed of different levels of cache to alleviate the penalties of accessing the main memory.

Considering the fact that the semiconductor technology is still capable of doubling the transistors on a chip every two years, it is clear that the number of cores on a chip will continue to increase. For instance, the Intel Tera-scale Computing Research Program has recently created a prototype chip containing 80 cores. In this context of many-core architectures, researchers are also exploring the use of 3D chip stacking to provide large, low-latency, last-level caches stacked on top of processors. This reduces power consumption and also improves bandwidth [HBK06].

Consequently, applications must evolve to efficiently exploit the potential of

multicore platforms. Old sequential applications must now be split into pieces (*e.g.*, tasks) that can be executed in parallel by threads, each one running on a specific core. The side effect is that the application data, which were accessed by a single thread on a sequential application, is now shared among several concurrent threads. Since usually applications are not embarrassingly parallel, it is necessary to use synchronization mechanisms to coordinate concurrent accesses to these shared data.

Traditional synchronization mechanisms such as *locks*, *mutexes* and *semaphores* have been extensively used to synchronize threads on multicore platforms and systems. They are simple to implement in hardware and they are safe mechanisms to deal with concurrent accesses to shared data. However, such simplicity comes with significant drawbacks. Firstly, they are considered as "low-level" mechanisms, since one must explicitly control the access of shared variables. Secondly, they cause blocking, so threads always have to wait until a lock (or a set of locks) is released. Thirdly, the careless use of such mechanisms can easily result in deadlocks or livelocks [Tai94]. Finally, blocking considerably limits scalability and adds complexity to the source code.

Due to the previously discussed issues, researchers have been looking for alternative mechanisms. One of such alternative mechanisms that has been subject of intense research in the last years is Transactional Memory (TM). The TM programming model allows programmers to write parallel portions of the code as transactions, which are guaranteed to execute atomically and in isolation regardless of eventual data races [Dal+10; HLR10]. At runtime, transactions are executed speculatively and the TM runtime system continuously keeps track of concurrent accesses and detects conflicts. Conflicts are then solved by re-executing conflicting transactions. Therefore, it removes from the programmer the burden of correct synchronization of threads and provides an efficient model for extracting parallelism from the applications.

Different implementations of TM systems make tradeoffs that impact both performance and programmability. The most common design choices are STM, Hardware Transactional Memory (HTM) and Hybrid Transactional Memory (HyTM). STM implements everything in software, so there is no need for a specific hardware [DGK09; DSS06; FFR08]. On the contrary, HTM implements all functionalities in hardware [McD+05; Moo+06]. The HyTM is a hybrid approach in which the hardware simply serves to optimize the performance of transactions that are controlled fundamentally

by software [Kum+06; Shr+06].

STM has some advantages over the other two approaches. It offers flexibility in implementing different mechanisms and conflict detection/resolution policies. It is easier to be modified or extended, it is not limited by small fixed-size hardware structures (such as cache memories) and it does not require specific hardware (thus it can be used on current platforms). Additionally, hardware and hybrid solutions are still in a premature stage and their implementation on commercial processors has just get started. The disadvantage is that STM performs worse than HTM and HyTM.

In general, the efficiency of parallel applications relies upon matching the behavior of the application with the underlying system and platform characteristics. This issue becomes much more complex in STM basically due to two reasons: (i) the TM model uses speculation, hence TM applications present an irregular behavior (data dependencies between threads are only known at runtime); and (ii) each STM system implements its own mechanisms to detect and solve conflicts and thus the same TM application can behave differently when the underlying STM system is changed.

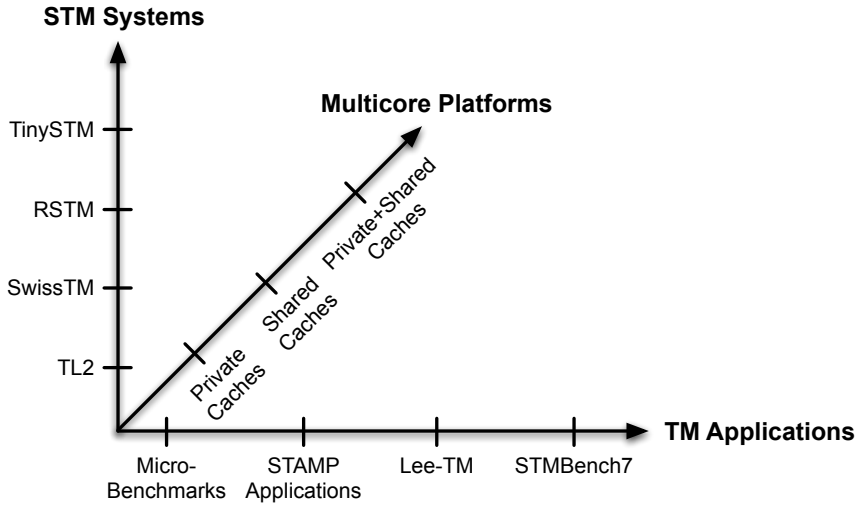


Figure 1.1: STM systems, TM applications and multicore platforms.

Figure 1.1 illustrates this scenario. On the *STM system*-axis, we included some of the STM systems currently available. Each one has its peculiarities such as the aforementioned mechanisms to detect and solve conflicts. On the *TM applications*-axis, we included the most known applications and benchmarks that are used to

evaluate STM systems. Some benchmarks such as the Stanford Transactional Applications for Multi-Processing (STAMP) are composed of several applications [Min+08]. TM applications may differ in several aspects such as the level of concurrency, the probability of conflicts, the size and the time spent inside transactions. Finally, on the *multicore platforms*-axis, we represented the differences in terms of memory hierarchy of multicore platforms: only private caches, only shared caches or mixes of both. All those aspects can have an important impact on the performance of applications.

In this thesis, we plan to **understand** and **improve** the performance of TM applications on multicore platforms. However, this is not trivial due to the several possible combinations of the previously cited aspects. We intend to analyze the performance and propose improvements to TM applications by using low-intrusive techniques to both TM applications and STM systems.

1.1 Research issues

As previously mentioned, the contributions of this thesis are situated around the analysis and improvement of the performance of TM applications on multicore platforms. In this section, we present some possible ways of exploring these issues and we define our main focus on each issue.

1.1.1 Understanding the performance of TM applications

The increased complexity in the development of parallel programs can be eased up by a good understanding of the effective application behavior in its specific hardware and software execution contexts [Lou+09]. Such information can be useful to understand and improve the performance of parallel applications. There are basically two main approaches to achieve this goal, *i.e.*, runtime analysis and post-mortem analysis.

- **Runtime analysis**: collects runtime information about the application behavior and uses such information to perform some action at runtime. Runtime analysis usually requires a small amount of storage and frequently relies on

sampling techniques to reduce the overhead [SBS11]. As an example, we can cite the work presented in [SH11], in which the authors use runtime analysis to characterize the workloads and adapt processor frequencies based on performance counter measures at runtime.

- **Post-mortem analysis**: the collected runtime information is recorded in a detailed log (trace file) for later analysis. The trace file is usually composed of timestamped *events* and their *attributes* [She99]. An event is typically represented by an ordered tuple that consists of the event identifier, the timestamp when the event occurred, where it occurred (*e.g.*, a thread identifier) and optional fields for specific information. The trace file can then be analyzed with a visualization tool such as Vampir [Knu+08].

In this thesis we explore both techniques to understand and improve the performance of TM applications. More precisely, we focus on techniques that do not add considerable intrusiveness to both TM applications and STM systems. This subject is further discussed in Chapters 3 and 4.

1.1.2 Improving the performance of TM applications

The performance of TM applications can be improved in several ways. In this section, our main objective is not to present an exhaustive list of works that aim at improving the performance of TM applications. Instead, we intend to show that the performance can be improved at different levels, ranging from the application to the platform levels. Then, we situate our approach proposed in this thesis.

- **TM application**: There are several ways of improving the performance at the TM application level. One possible approach is to change the TM application source code in such a way that the probability of having conflicts is reduced. One of such techniques is called *privatization*, which temporarily privatizes a shared data during a computation to reduce conflicts [MSS08]. When conflicts are reduced, there will be less transactions being re-executed due to conflicts. Consequently, the application will take less time to execute.

- **STM system**: Another alternative to improve the overall performance of a TM application is modifying the STM system algorithms to fit the application workload. For instance, some STM systems present better performance with read-dominant workloads whereas others are more suitable for write-dominant workloads. In [MIS05], the authors proposed an Adaptive STM system (ASTM) that automatically adapts its behavior to the application workload, allowing it to closely approximate the performance of the best existing system for that workload.

- **Operating system**: Current mainstream operating systems do not have any integration with Transactional Memory. The performance of TM applications can be enhanced if operating systems offer special integration with HTM as proposed in [Ros+07]. Another example of optimizations at the operating system level is proposed in [Mal+11]. In this work, the authors extended the Linux scheduler to support deadlines for reactive TM applications.

- **Platform**: At the platform level, we can consider two possible types of optimizations. On the one hand, it is possible to construct a specific platform that implements TM in hardware to accelerate the execution of TM applications. As an example, we can cite the Intel's next generation processor microarchitecture named *Haswell*[1], which will provide instruction set extensions that allow programmers to specify regions of code for transactional synchronization. On the other hand, TM applications can better exploit the full potential of current multicore platforms by taking into consideration the platform characteristics such as the memory hierarchy [WL10]. In this case, the performance gains can be obtained by matching the characteristics of the TM application to those of the underlying platform.

We believe that the performance of TM applications can be improved if we match its characteristics (along with the characteristics of the STM system) to the underlying multicore platform. More precisely, we propose to gather these characteristics from the TM applications and STM systems and then use such information to better exploit the memory hierarchy of modern multicore platforms.

[1] More information available at http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/

One technique to deal with that is called thread or processes mapping, which aims at mapping threads or processes to specific cores to improve the use of resources such as interconnections and cache memories. Although the impacts of applying thread mapping on TM applications has not been explored yet, we believe that it could be beneficial due to some intuitions. For instance, consider a TM application in which transactions constantly access the same amount of shared data. In this case, placing threads on sibling cores may reduce the latency because the data will probably be stored into the cache shared by these threads. This may be the case of high-conflicting TM applications. Now, consider an opposite case where a TM application is composed of several transactions that usually access a large amount of disjoint data, thus rarely conflicting. In this case, distributing threads across different processors (thus avoid cache sharing) may reduce the contention on the same cache, making more cache available for each thread.

In this thesis we want to confirm these intuitions and propose an approach capable of predicting suitable thread mappings for TM applications to improve their performances. This subject is discussed in more details in Chapter 4.

1.2 Contributions

The first contribution of this thesis concerns the comprehension of the performance of TM on multicore platforms. In order to do that, we first take a deeper look on the impacts of STM systems on the performance of TM applications. We show that the performances of applications using TM-based synchronization solutions depend on both the applications themselves and the STM system specifics. We support this fact by demonstrating that the use of TM may result in worse or better performance for the application, depending on these specifics. In order to gain some insight on these issues, helping developers to understand and improve their performance, we propose a generic approach for collecting and tracing relevant information about transactions. Our solution can be applied to different STM systems and applications as it does not modify neither the target application nor the STM system source codes. We then show that the collected information can be helpful in order to comprehend the performance of TM applications.

This joint work with Kiril Georgiev (Ph.D. student at *Université de Grenoble*),

Vania Marangozova-Martin (assistant professor at *Université de Grenoble*) and Miguel Santana (head of the Embedded Software Development Tools department in STMicroelectronics Central R&D group) resulted in a publication in the proceedings of the Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP) in 2011:

- Márcio Castro, Kiril Georgiev, Vania Marangonzova-Martin, Jean-François Méhaut, Luiz Gustavo Fernandes, and Miguel Santana. "Analysis and Tracing of Applications Based on Software Transactional Memory on Multicore Architectures". In: *Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)*. Aya Napa, Cyprus: IEEE Computer Society, 2011, pp. 199–206. ISBN: 978-0-7695-4328-4. DOI: 10.1109/PDP.2011.27

The second contribution of this thesis concerns the proposal of an approach to improve the performance of TM applications through the exploitation of the memory hierarchy of modern multicore platforms. We evaluate and demonstrate that the use of thread mapping strategies allows us to make better use of the underlying multicore platform and thus improve the performance of TM applications. However, the efficiency of such approach relies upon matching the application behavior with system characteristics. Particularly, STM systems make this task even more difficult due to its runtime system. Existing STM systems implement several conflict detection and resolution mechanisms, which leads TM applications to behave differently for each combination of these mechanisms. Thus, the prediction of a suitable thread mapping strategy for a specific application/STM system becomes a daunting task. We tackle this problem by using Machine Learning (ML) to automatically infer a suitable thread mapping strategy for TM applications. Our approach takes into account not only the characteristics of the TM application but also the STM system and the underlying multicore platform.

This joint work with Luís Fabrício Góes (Ph.D. student at University of Edinburgh), Prof. Murray Cole and Prof. Marcelo Cintra (both from the University of Edinburgh) resulted in the following publication in the proceedings of the High Performance Computing Conference (HiPC) in 2011:

- Márcio Castro, Luís Fabricio Wanderley Góes, Christiane Pousa Ribeiro, Murray Cole, Marcelo Cintra, and Jean-François Méhaut. "A Machine Learning-Based Approach for Thread Mapping on Transactional Memory Applications". In: *High Performance Computing Conference (HiPC)*. Bangalore, India: IEEE Computer Society, 2011, pp. 1–10. ISBN: 978-1-4577-1949-3. DOI: 10.1109/HiPC.2011.6152736

Finally, our third contribution extends the aforementioned approach to predict and apply suitable thread mapping strategies for TM applications in a dynamic fashion. We argue that more complex applications will make use of TM in a near future. Those applications can be composed of multiple execution phases with a potentially different transactional behavior in each phase. Thus, instead of predicting and applying a single static thread mapping strategy, we use profiling techniques to gather useful information during the execution of TM applications, switching the thread mapping strategy to a more adequate one at runtime when necessary. We implemented this approach in a state-of-the-art STM system, making it transparent to the user.

The preliminary results of our third contribution were presented and discussed in the Euro-TM Workshop on Transactional Memory (WTM) in 2012 (co-located with EuroSys 2012). The workshop consisted of short presentations and did not have any published proceedings to facilitate later submission to other venues. The submissions were evaluated by members of the Euro-TM Management Committee.

- Márcio Castro, Luís Fabrício Góes, Luiz Gustavo Fernandes, and Jean-François Méhaut. "Dynamic Thread Mapping Based on Machine Learning for Transactional Memory Applications". In: *Euro-TM Workshop on Transactional Memory (WTM)*. Extended abstract. Apr. 2012

An extended version of this work was accepted for inclusion in the technical program and the proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par) in 2012. This was a joint work with Luís Fabrício Góes (Ph.D. student at University of Edinburgh).

- Márcio Castro, Luís Fabrício Góes, Luiz Gustavo Fernandes, and Jean-François Méhaut. "Dynamic Thread Mapping Based on Machine Learning for Transac-

tional Memory Applications". In: *International European Conference on Parallel and Distributed Computing (Euro-Par)*. Vol. 7484. Lecture Notes in Computer Science (LNCS). Rhodes Island, Greece: Springer-Verlag, 2012, pp. 465–476. ISBN: 978-3-642-32819-0. DOI: 10.1007/978-3-642-32820-6_47

1.3 Scientific context of the thesis

This thesis was funded by a subproject of Nano 2012 program called OPM^2: Analysis and Observation of Multithreaded Applications on Multicore Processors. In the context of this thesis, one of our main contributions to this project was to evaluate and analyze the performance of Transactional Memory on multicore platforms [CD10; Cas10; Cas+10].

The research that led to the published papers cited in the previous section as well as all the contents of this thesis was carried out in the **Nanosim** team. Nanosim stands for *Nanosimulations and Embedded Applications for Hybrid Multi-core Architectures* and it is one of the research teams of the Grenoble Informatics Laboratory (*Laboratoire d'Informatique de Grenoble* - LIG). The main research fields of Nanosim are High Performance Computing (HPC) and Embedded Systems. More precisely, in the field of HPC, the Nanosim team is interested in providing environments to better exploit multiprocessor architectures. Nanosim members have also straight collaborations with Brazilian universities such as the *Universidade Federal do Rio Grande do Sul* (UFRGS), the *Universidade Federal de São Paulo* (USP), the *Pontifícia Universidade Católica do Rio Grande do Sul* (PUCRS) and the *Pontifícia Universidade Católica de Minas Gerais* (PUC Minas).

During this thesis, I have developed other scientific works in collaboration with members from Nanosim and other research teams. Some of these collaborations are closely related to this thesis whereas others are related to high performance computing on parallel platforms in general. These collaborations began when I was a master's student in Brazil under the direction of Prof. Luiz Gustavo Fernandes. My master's thesis was about the parallelization of a Geophysics application that exploited memory affinity strategies for Non-Uniform Memory Access (NUMA) platforms. This work was closely related to the thesis of Christiane Pousa Ribeiro (Nanosim, *Université de Grenoble*). At that time, Christiane was under the direction

of Jean-François Méhaut (*Université de Grenoble*) and her main research topic was *memory affinity for NUMA platforms* [Rib11]. Such common research topic allowed us to collaborate during my thesis and resulted in several publications in national and international conferences [Cas+09; Rib+10; Rib+09b; Rib+09a]. All these works aimed at using memory affinity strategies to improve the performance of high performance scientific applications.

More recently, I also collaborated with Prof. Henrique Cota de Freitas and Carlos Augusto Paiva da Silva Martins, both professors at *Pontifícia Universidade Católica de Minas Gerais* (PUC Minas), Brazil. These works concerned the analysis and evaluation of parallel workloads on multicore platforms and also led to publications in international conferences [Oli+11; Rib+11].

1.4 Thesis outline

The remaining chapters of this thesis are organized as follows:

- Chapter 2 reviews the basic concepts and topics that are relevant throughout this thesis. We briefly discuss the evolution of multicore platforms as well as their impacts on the performance of parallel applications, we present an overview of Transactional Memory along with some important design criteria that impact its performance and we describe the most known applications and benchmarks used to evaluate Transactional Memory systems.

- Chapter 3 concerns the comprehension of the performance of TM applications on multicore platforms. We analyze the impacts of both applications and STM systems on the overall performance and propose a generic approach for collecting relevant information from TM applications. We then use such approach to gain some insights about the performance of TM applications.

- Chapter 4 proposes the use of Machine Learning to improve the performance of TM applications through the exploitation of the memory hierarchy of multicore platforms. Our approach considers the characteristics of the TM application, STM system and platform to infer an efficient thread mapping adapted to the underlying environment.

- Chapter 5 presents an experimental evaluation of our ML-based approach for thread mapping on TM applications on two multicore platforms.
- Chapter 6 discusses several related works concerning the performance evaluation of TM systems and applications, the use of thread mapping to improve the performance of applications and the use of machine learning to construct heuristics for improving the performance of parallel applications.
- Chapter 7 presents our conclusions and perspectives.

CHAPTER 2

Background

THIS chapter aims at reviewing the basic concepts and topics that are relevant throughout this thesis. More precisely, in Section 2.1, we briefly discuss the evolution of multicore platforms as well as their impacts on the performance of parallel applications. In Section 2.2, we present an overview of Transactional Memory along with some important design criteria that impact its performance. In Section 2.3, we describe the most known applications and benchmarks used to evaluate Transactional Memory systems. We then conclude this chapter in Section 2.4.

2.1 Multicore platforms

Historically, processor manufacturers have responded to the demand for more processing power by delivering faster processor speeds. However, the need to achieve higher performance without driving up power consumption and heat has become a critical concern. The solution for this problem is based on the fact that a good overall processing performance can be achieved by reducing individual core clock speeds while increasing the number of cores. Consequently, this can lower the heat of individual cores and increase the overall performance.

2.1.1 Architectural concept

Although the organization of a multicore platform can vary depending on manufacturer and product development over time, they share some basic features: processors have two or more processing units (cores) and cache modules. These cache modules are arranged hierarchically and can either be shared or independent. Usually, cache memories closer to the cores tend to be private whereas higher level caches tend to be shared.

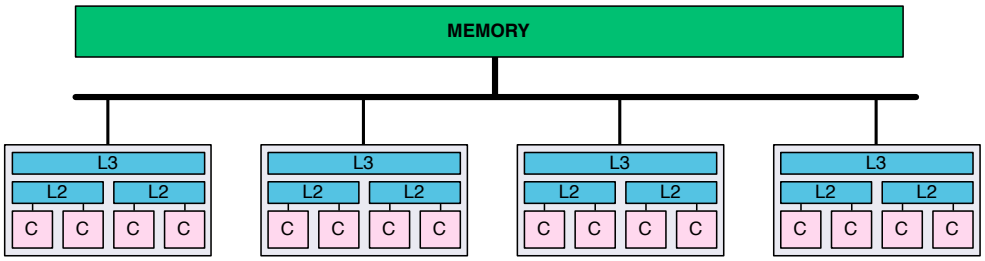


Figure 2.1: Example of a multicore platform with two levels of shared caches.

Figure 2.1 illustrates a multicore platform composed of four quad-core processors. In this example, there are two levels of cache memories (L2 and L3) and a shared main memory. Since the main memory is connected to all processors through a single bus, the time spent to access a data on the main memory is uniform. This design is known as Uniform Memory Access (UMA). Another possible design is to use multiple memory banks physically distributed through the platform. In this case, the main memory still has a unique address space but the time spent to access data is conditioned by the distance between the processor and the memory bank in which the data is physically allocated. This design is called Non-Uniform Memory Access (NUMA).

2.1.2 Performance of multicores

It is well known that the rate of improvement in processor speed exceeds the rate of improvement in Dynamic Random Access Memory (DRAM) speed. Indeed, the performance of processors and DRAMs are improving exponentially, but the exponent for processors is substantially larger than that for DRAMs. This growing disparity leads to the so-called “memory wall problem” [McK04]. Obviously, the effective

performance that can be obtained from a multicore platform also depends on several other factors such as memory hierarchy, application, operating system and compiler optimizations.

Figure 2.2 illustrates this problem. In this example, we consider a multicore platform composed of two dual-core processors. According to the processor specification, each core has a theoretical performance of 10.6 GFlops[1] as shown in Figure 2.2a. This means that, in theory, this multicore platform would be capable of executing 42.4 GFlops. However, real world performance may not come very close to the theoretical value. To evaluate how far is the effective performance from the theoretical one, we performed a series of experiments with the parallel version of the LINPACK benchmark[2]. LINPACK is benchmark that solves a dense system of linear equations.

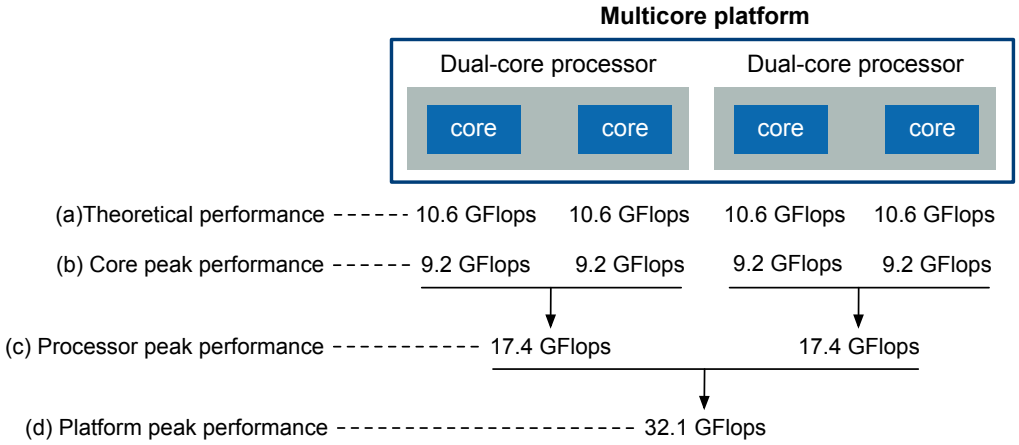


Figure 2.2: Example of the scalability potential of multicore processors.

We executed this benchmark with a single thread and obtained a peak performance of 9.2 GFlops (Figure 2.2b), which is lower then the theoretical core performance. Secondly, we executed the benchmark with two threads in the same processor, obtaining a peak performance of 17.4 GFlops (Figure 2.2c), which is inferior to what would be expected (*i.e.*, 2×9.2 GFlops). A similar behavior happened when we executed the benchmark with four threads. In this case, we obtained a peak effective performance of 32.1 GFlops (Figure 2.2d), which is inferior to 2×17.4

[1] GFlops stands for "billions of floating-point operations per second".
[2] LINPACK can be download at http://www.netlib.org/linpack/

GFlops and far away from the theoretical performance of the platform (*i.e.*, 42.4 GFlops).

This example must be considered for demonstration purposes only. Since LINPACK is cpu-bound and solves a very regular problem, the performance achieved is quite high and gives a good estimation of the peak performance of the platform. However, the effective performance depends on several factors. One of such factors is the memory hierarchy, which can considerably impact the overall performance of applications that make intensive use of memory (memory-bound applications).

2.1.3 Impacts of the memory hierarchy

In the context of memory hierarchy, as the distance between the core and the memory increases, the time needed to access a data physically stored on such memory also increases. This means that cores have a fast access to lower cache levels such as L1 and L2 than higher memory levels such as the main physical memory. Considering this fact, one can realize that it is always better to place threads on sibling cores. This can reduce the memory access latency by sharing all levels of the cache hierarchy, allowing threads to reuse the data that is already on cache. Indeed, this strategy can be used to improve the performance of some applications but it is not suitable for all applications. For instance, consider a memory-bound multithreaded application where threads usually access disjoint data. In this case, it may be better to distribute threads across different processors in such a way that they will not share caches to alleviate contention.

We demonstrate this scenario with a synthetic parallel application. As its core, the application is very simple: each thread executes a `for loop` with a fixed number of iterations and then exits. Each iteration consists of a fixed number of read and write operations on a one-dimension array of integers. Although the order of read and write operations as well as the array indexes to be accessed are chosen randomly, the algorithm guarantees the number of read and write operations as specified in the input parameters. The number of iterations, threads and the size of the array can also be specified.

We made two variations of this synthetic parallel application. The first one, named *shared array*, has a single shared array that is accessed by all threads. Since

we are only interested on the impact of the memory hierarchy, we do not use any synchronization mechanism to guarantee consistent results. The second one, named *private arrays*, has t private arrays, where t corresponds to the number of threads. In this case, each thread accesses only its private array.

In order to perform our experiments, we fixed the input parameters of the synthetic application as follows: the number of threads was fixed to 8, each thread executes 100,000 iterations and each iteration performs 1024 reads and 1024 writes. In the shared array variation, we used a single shared array of size 65,536. In the private arrays variation, on the other hand, each thread has a private array of size 65,536.

We then carried out several experiments with both variations of the synthetic application using the previously described input values. The target machine was a multicore NUMA platform composed of four 8-core Intel Xeon X7560 at 2.27GHz. Each core has private L1 (32KB) and L2 (256KB) caches and each processors has a shared L3 cache (24MB). During the experiments, we fixed the threads to the cores following two distinct strategies: one that pins all threads on the same processor and other that spreads threads among different processors. The goal is to observe the impact of the cache hierarchy on the performance of the synthetic application. The results are shown in Figure 2.3.

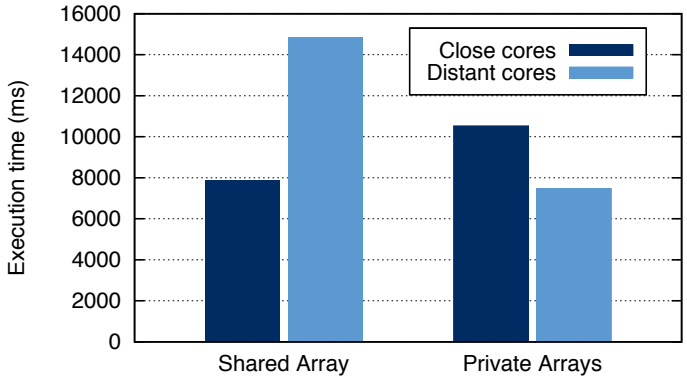


Figure 2.3: Example of the impacts of memory hierarchy on the performance of a synthetic parallel application.

As it can be noticed, the performance of the synthetic application changes when we change the way threads use the cache hierarchy. The shared array variation

has better performance when threads are placed on sibling cores, since most of the shared data is on cache, reducing the latency. On the contrary, the use of different caches improved the performance of the private arrays variation, since this scheme reduces cache pollution and contention.

2.1.4 Synchronization of shared data

To efficiently exploit the full potential of multicore platforms, applications must evolve. Old sequential applications are now parallel: they are split into pieces that are executed in parallel by threads, each one running on a specific core. The side effect is that the application data, which were accessed by a single thread on a sequential application, is now shared among several concurrent threads.

In a multithreaded environment, if multiple threads access the same resource for read and write, the value may not be the correct value. For instance, consider an application that contains two threads, one thread for reading the content from a file and another thread writing the content to the file. If the write thread tries to write and the read thread tries to read the same data, the data might become corrupted. One possible solution for this problem is to *lock* the file when the writer thread is modifying the file content. This means that the read thread will have to wait until the lock is released to proceed.

Indeed, the correct synchronization of multiple accesses to shared data is challenging. Depending on the complexity of problem that is parallelized, more or less mechanisms must be used to guarantee the correct execution of a parallel application without decreasing its scalability.

Among all possible synchronization mechanisms, locks are the most commonly used. A mutually exclusive (or **mutex**) lock is used to protect a **critical section**, which is a segment of code that only one thread at a time is allowed access. Differently, a **semaphore** can grant access to one or a limited number of threads. Some languages have a slightly higher level construct, a **monitor**, to prevent concurrent accesses.

The above mechanisms are considered as “low-level” mechanisms, since one must explicitly control the access of shared variables. They cause blocking, so threads always have to wait until a lock (or a set of locks) is released. This may add considerable overhead **even when the chances for collision are very rare**. In some

cases, although the chances for collision are very rare, the accesses to shared data must be protected if we want to avoid race conditions, thus limiting the scalability of the application [HLR10; HX98].

We demonstrate how locks can limit the scalability by using the well-known Traveling Salesman Problem (TSP) [App+07], which finds the shortest possible path between two nodes in a graph by visiting each node exactly once. In our parallel implementation, the graph exploration is done by multiple threads. Concurrent threads access different shared data, such as the current shortest path and the pool of paths to explore. To avoid race conditions, we used mutex locks to protect concurrent accesses to shared variables. This was the case of the shared variable `minimum`, which stores the current shortest distance.

We implemented two variations of TSP. Our first approach is based on the fact that all global shared variables that are accessed concurrently should be protected with mutex locks (named *strict*). For the second approach, after a carefully analysis of the source code, we removed some of the mutex locks without creating data races (named *relaxed*). That was the case of some read-only accesses to the variable `minimum`, which can be relaxed without breaking the application correctness. Figure 2.4 shows an example of portions of the source code from the strict and relaxed versions, where the synchronization has been removed.

```
lock (lock_minimum);
if length ≥ minimum then
    (*cuts)++;
    unlock (lock_minimum);
    return 1;
end
unlock (lock_minimum);
```

(a) strict

```
if length ≥ minimum then
    (*cuts)++;
    return 1;
end
```

(b) relaxed

Figure 2.4: Snippets of the strict (left) and relaxed (right) versions of TSP.

This specific portion of code is executed at the beginning of the main recursive function of TSP and avoids the successive recursive calls to the main function when a thread finds a path that is larger than the global shortest path. The reason for

allowing the mutex lock to be removed in this portion of code without creating data races is twofold. Firstly, concurrent threads only access this variable for read-only operations in this portion of code. Secondly, this variable can also be updated by concurrent threads (and those accesses are still protected by mutex locks) if a shorter path is found. However, when this occurs, the new value to be written will be inevitably lower than before. Thus, it does not invalidate previous results, since those previous paths continue to be greater or equal to this new value.

The execution times (in log scale) of these two approaches are shown in Figure 2.5. Results represent arithmetic means of 30 executions on the SMP-24: an SMP multicore platform based on four six-core Intel Xeon X7460 (we present a more detailed description of this platform in Section 5.1.1). In all experiments, we used the same seed for the pseudorandom number generator and a graph composed of 16 nodes.

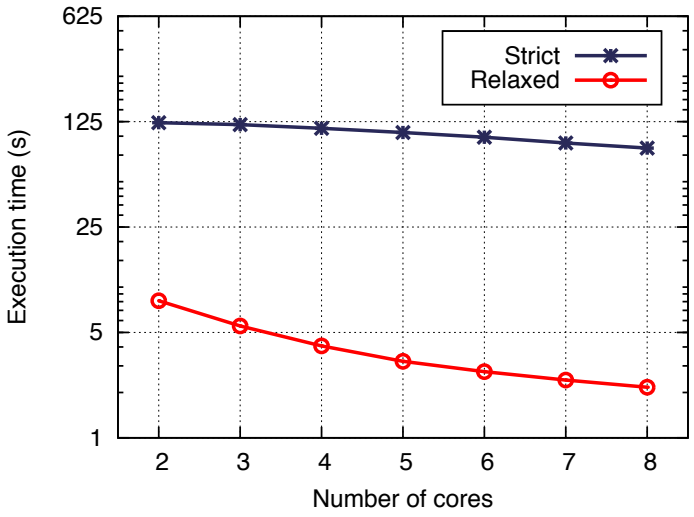


Figure 2.5: Execution times of the two variations of TSP (strict vs. relaxed).

Considering the first approach, the results show high execution times and poor scalability. This occurs due to the large number of accesses to the shared variable `minimum`, causing threads to be blocked continuously. When we relaxed some synchronization on read-only accesses to this variable the performance was considerably increased. This is a simple example but it can be very hard to be sure that removing a lock to increase the application scalability will not break data consistency.

Due to all these issues concerning those traditional mechanisms, researchers are looking for alternative synchronization mechanisms. One of those that is attracting considerable attention of researchers is Transactional Memory.

2.2 Transactional memory

The Transactional Memory programming model offers a new attractive way of developing parallel applications using a higher abstraction level. It shifts the problem of correct synchronization to the TM system, which is responsible for making sure that deadlocks will not occur, race conditions are correctly handled and locks are performed at a granularity which allows to indeed exploit the inherent parallelism of the application.

The original idea dates back to 1977, when *D. Lomet* realized that an abstraction similar to a database transaction might make a good programming language mechanism to ensure the consistency of data shared among several processes [Lom77]. Sixteen years later, in 1993, *M. Herlih* and *J. Moss* proposed a hardware-supported Transactional Memory as a mechanism for building lock-free data structures [HM93]. Since then, there has been a growing interest of researches on Transactional Memory.

This section aims at bringing up some important aspects of Transactional Memory. First, we present its general concepts. Then, we discuss the different design choices and approaches for implementing Transactional Memory. Finally, since we are interested in Software Transactional Memory in this thesis, we present a more detailed description of its peculiarities and we briefly describe the most known state-of-the-art STM systems.

2.2.1 General concepts

The basic idea behind Transaction Memory comes from Transactional Database Management Systems (DBMS), in which a transaction is a sequence of actions that appears indivisible and instantaneous to an outside observer [GUW08]. In these systems, two or more queries conflict when different transactions perform read and write instructions over a database in such a way that the result could not arise from a sequential execution of the queries. In this context, transactions ensure that all

queries produce the same result as if they executed serially. A database transaction enforces some properties called **ACID**: atomicity, consistency, isolation and durability.

Atomicity refers to the ability of the DBMS to guarantee that either all tasks of a transaction are performed or none of them is performed. It is not acceptable for a constituent action to fail and for the transaction to finish successfully nor it is acceptable for a failed action to leave behind evidence that it executed [HLR10]. Thus, there are two possibilities for an executing transaction: it can be either **committed** (if it completes successfully) or **aborted** (if it fails).

Another important property is the **isolation**, which refers to the requirement that other operations cannot access (or see) the data in an intermediate state during a given transaction. Because of that, transactions must produce a correct result, regardless of which other transactions are executing concurrently.

The next property of a transaction is **consistency**. This property ensures that the database remains in a consistent state before starting a transaction and after finishing it (whether successful or not). Any data written to the database must be valid according to all defined rules, such as integrity constraints, cascades and triggers.

The final property is **durability**, which requires that once a transaction commits, its result must be permanent and available to subsequent transactions even in case of system failures. Many databases implement durability by writing all transactions into a transaction log that can be played back to recreate the system state right after a system failure.

It is important to mention that TM systems usually do not provide durability and consistency properties [LK08]. In general, TM systems assume that changes in memory need not be durable (mainly if the underlying system is not a persistent one) and they do not consider the previously mentioned consistency rules while modifying data inside transactions.

The basic concepts of Transactional Memory are illustrated in Figure 2.6. Transactions are sequences of steps (delimited by a blocks of code) executed by threads [HS08]. The way a transaction is defined inside the source code depends on the implementation. However, most compilers that support TM provide a simple **atomic** statement, which is responsible for executing the inner block of code (and the routines it invokes) as a transaction.

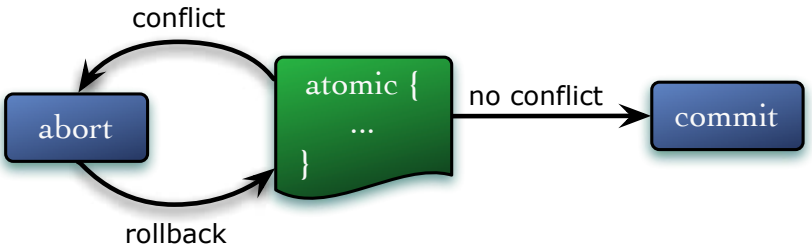


Figure 2.6: Transactional Memory basic concepts.

The portion of code written inside the atomic block is guaranteed to be executed atomically and in isolation regardless of eventual data races [Dal+10; HLR10]. At runtime, transactions are executed speculatively and the Transactional Memory continuously keeps track of concurrent accesses and detects conflicts. When a conflict arises, only one transaction involved in the conflict will **commit** whereas the others will be **aborted** and then re-executed (this action is also called *rollback*). When there is no conflict, all transactions are allowed to commit simultaneously.

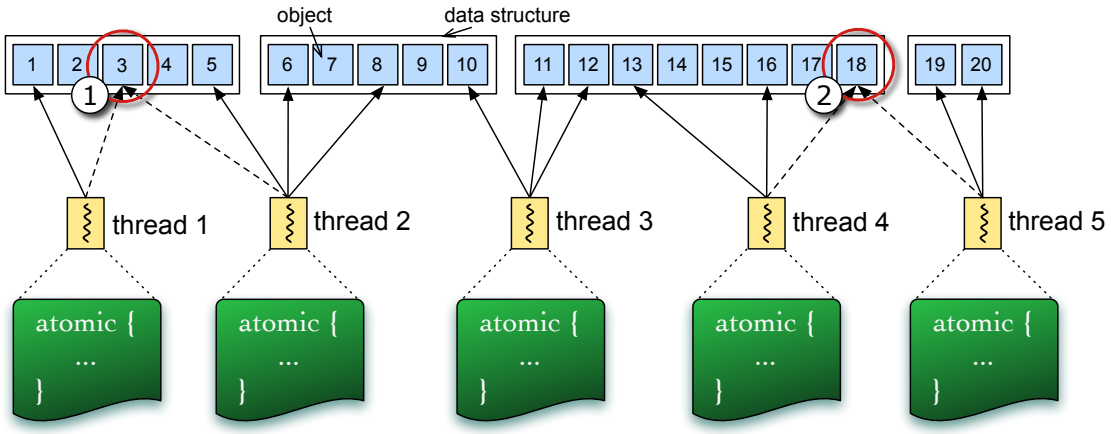


Figure 2.7: Example of conflicting and non-conflicting scenarios.

Figure 2.7 illustrates the use of Transactional Memory for guaranteeing atomicity while accessing shared data structures. In this example, there are five threads that read/write values from/to the objects of four shared structures (memory accesses are represented by arrows). In order to guarantee "atomic" accesses, each thread encapsulates all of its operations inside transactions (represented by the atomic

blocks). To simplify the example, we consider an object-based TM system, which can detect conflicts between transactions at object granularity. This means that conflicts may only occur when two or more transactions are accessing the same object at the same time and at least one transaction is modifying this object. We discuss other designs in more detail in Section 2.2.2.

As it can be noticed, there are some threads that access the same object. These accesses are represented by dashed arrows. More precisely, threads 1 and 2 access object **3** simultaneously (Scenario 1) whereas threads 4 and 5 access object **18** (Scenario 2). Let us consider these two different scenarios as read and write operations chronologically represented as follows:

- **Scenario 1**: no conflict (read-only transactions)
  1. *thread 1* reads the objects [1, **3**] and performs some computation;
  2. *thread 2* reads the objects [**3**, 5, 6, 8] and performs some computation;
  3. *thread 2* commits;
  4. *thread 1* commits.
- **Scenario 2**: read-write conflict
  1. *thread 4* reads the objects [13, 16, **18**] and performs some computation;
  2. *thread 5* reads the objects [**18**, 19, 20] and performs some computation;
  3. *thread 5* updates the object **18** (write operation);
  4. *thread 4* aborts and rollbacks;
  5. *thread 5* commits.

In scenario 1, there are two read-only transactions accessing the same shared object **3** at the same time. In this case, the Transactional Memory mechanism can commit both transactions since they do not conflict.

On the contrary, a different situation arises in scenario 2. In this case, there is a conflict due to the fact that *thread 4* reads object **18** which is further updated by the transaction executed in *thread 5*. This conflict is solved automatically by the Transactional Memory, so the developer does not need to take care of it. One possible solution to solve such conflict is shown in steps 4 and 5. In this example,

the write transaction in *thread 5* was committed and read-only transaction in *thread 4* was aborted. Thus, the transaction in *thread 4* will be re-executed and will then use the new value updated by *thread 5*. The decision on what transaction to abort/commit can be different depending on the TM implementation. There exist different mechanisms to solve conflicts between transactions and each one has its pros and cons. We discuss some of the most used mechanisms later on.

It is important to notice in this example that some transactions also access disjoint elements of the shared structures without blocking on the shared structures. If locks were used to protect those concurrent accesses, it would be necessary to use one lock per shared element to obtain such fine granularity. Additionally, one would have to be very careful to not create deadlocks or livelocks. For instance, consider that two threads are trying to acquire two locks (*e.g.*, $l1$ and $l2$) to modify two elements of the shared structure atomically. A deadlock will happen if *thread 1* acquires $l1$ and blocks on $l2$ due to the fact that *thread 2* has already acquired $l2$ and is still blocked on $l1$. This problem can be solved by imposing a total order to acquire the locks. However, the problem is that it could limit the program scalability and will add complexity to the source code. These kind of problems are avoided by the Transactional Memory.

Although the Transactional Memory promises to simplify the development of parallel applications, it has also some limitations. For instance, transactions by themselves cannot replace all synchronization mechanisms in parallel programs [ML06]. Beyond mutual exclusion, synchronization is often used to coordinate independent tasks. In this context, Transactional Memory provides little assistance in coordinating independent tasks [LK08]. Forcing a thread to wait until a specific task is finished or limiting the number of threads performing a specific task are examples of such coordination mechanisms that are not implemented by the TM model.

2.2.2 Design choices

Basically, there are four important criteria that must be taken into account while designing Transactional Memory systems: transaction granularity, version management, conflict detection and conflict resolution.

Transaction granularity is the unit of storage in which a TM system detects conflicts [HLR10]. For object-based languages, it is common to use **object granularity**,

which detects conflicts when threads modify the state of objects. There are also other alternatives such as **word granularity** and **block granularity**. The former uses a memory word as the unit of conflict detection whereas the latter uses a group (block) of words. The chosen granularity can obviously impact the TM performance and it also influences the way it will be implemented by software and/or hardware.

The second important mechanism is **version management**. Since a transaction typically modifies data in memory, it is important to control how these modifications are managed. In general, TM systems use one of these two possible approaches: **direct update** (also known as *eager version management*) and **deferred update** (also known as *lazy version management*). If direct update is applied, the transaction directly modifies the data and the system uses some sort of concurrency control to prevent other transactions from concurrently modifying the object or committing after reading an old value. Direct update requires that the system record the original value, so it can be restored in case of transaction abortion. On the other hand, if deferred update is used, the data is modified in a private location and also read from there. This mechanism allows other transactions to modify their private data concurrently. When a transaction commits, it updates the data using the private copy. In case of abort, the transaction just discards this private copy.

The third important mechanism is **conflict detection**. Basically, a conflict can be detected as soon as it happens or postponed until a transaction commits. In the first approach, called *eager conflict detection*, read/write conflicts are detected as they occur during the transaction execution. In contrast, if *lazy conflict detection* is used, conflicts are only detected at commit-time. This reduces the overhead during the transaction execution at the cost of increasing the overhead on a commit operation.

As soon as a conflict is detected, a **conflict resolution** mechanism is invoked. There are many different algorithms that are used to select which transactions must be aborted in order to guarantee forward progress of the program. This task is usually addressed by the **contention manager**, which implements one or more **contention resolution policies** which determine whether conflicting transactions should abort, wait or proceed. Two common alternatives are to abort one of the conflicting transactions immediately (called suicide) or wait for a random, exponentially increasing, delay before restarting (called backoff). Obviously, the choice of which transaction must be aborted can also affect the performance of the system.

2.2.3 Implementation approaches

The aforementioned design choices are the core of Transactional Memory. However, they can be implemented in three different ways: only in software, only in hardware or both. In the next sections, we discuss this subject, evaluating the pros and cons of each approach.

Software Transactional Memory

A Software Transactional Memory (STM) system implements all the transactional semantics in software. The advantage of STM for system programmers is that it offers flexibility in implementing different mechanisms and policies at the software level. Moreover, software is easier to modify and has fewer intrinsic limitations imposed by fixed-size hardware structures (*e.g.*, caches). There is also an important advantage for end users, since they can port their applications for Transactional Memory without needing extra specific hardware [HLR10].

However, STM has also some drawbacks, mainly with respect to performance. Since everything is implemented in software, STM usually presents higher overheads than traditional synchronization mechanisms. More precisely, this overhead is more evident when the application is running with low thread counts [Wam+12].

The performance of STM systems also depends on the workload. In general, STM systems present more expensive overheads than locks when a small number of processors is used. However, as the number of processors increases, the contention for a lock and the cost of locking also increase. When this occurs and conflicts are rare, STM systems have been showing comparable or even better results when compared to traditional locks [LK08]. We discuss this subject in more detail in Section 3.1.

In this thesis, we are interested in STM since it does not imply any specific hardware. We give a brief description of the most known state-of-the-art STM systems later on.

Hardware Transactional Memory

Contrary to the STM solution, Hardware Transactional Memory (HTM) implements all transactional functionalities in hardware (it manages data versions and

tracks conflicts transparently). The basic idea behind HTM is to modify the modern cache-coherence protocols in order to implement transactions (in fact, they already detect and resolve synchronization conflicts between writers as well as between readers and writers).

Since the mechanisms are implemented in hardware, HTM systems intend to have a better performance when compared to STM systems. However, HTM faces several system challenges that are not an issue for STM implementations. The caches used to track the read set, write set and data versions have finite capacity and may overflow during a long transaction. Thus, many HTM systems make assumptions on hardware, such as maximum transactions size and Operational System support.

The interest in full hardware implementation of TM dates to the initial two papers on TM by *T. Knight* [Kni86] and *M. Herlihy* and *J. Moss* [HM93]. HTM systems usually do not require software instrumentation of memory references within transaction code.

A more recent example of HTM is the Log-based Transactional Memory (LogTM). It is a HTM system that makes commits fast by storing old values to a per-thread log in cacheable virtual memory and storing new values in place [Moo+06]. It extends the directory-based MOESI cache coherence protocol to enable both fast conflict detection and fast commit. The coherence protocol tracks addresses accessed inside transactions and participates in conflict detection. An improvement of LogTM called LogTM-SE was proposed in [Yen+07] to support thread suspension and migration.

Hybrid Transactional Memory

Even though recent STM systems scale considerably well, the overhead of the software systems can be significant [LK08]. On the contrary, HTM systems present better performance than STM systems but they have several limitations imposed by hardware. In this context, it emerges the Hybrid Transactional Memory (HyTM), which intends to tackle these issues.

The primary source of overhead for an STM system is the maintenance and validation of read sets. In few words, a transaction T can commit (or continue to execute) successfully if it meets two conditions: (i) no other concurrently executing transaction has modified some data that has been read by T and (ii) transaction

T is not modifying some data that another transaction is also modifying. To do so, the STM system tracks the data that has been read/written inside transactions and validates it at commit-time. This operations may have a high cost when many transactions modify a large amount of data.

In order to address the limitation of hardware resources (*e.g.*, cache capacity), a transaction can start in the HTM mode using hardware mechanisms for conflict detection and version management. If HTM resources are exceeded, the transaction is rolled back and restarted in STM mode with additional instrumentation. However, the main challenge in such hybrid systems is to detect conflicts between transactions started in HTM mode and transactions started in STM mode.

In [Kum+06], the authors describe a HyTM solution that uses a hardware mechanism as long as transactions do not exceed resource limits and falls back to a software mechanism when those limits are exceeded. This approach combines the performance benefits of a pure hardware scheme with the flexibility of a pure software scheme. The hardware mechanism tracks transactionally accessed data at the cache line granularity, while the software scheme tracks it at the object granularity. It uses a simple strategy to choose between HTM and STM: if the transaction fails to commit successfully within three attempts in HTM mode, it falls back into STM mode and retries until it succeeds.

Differently, in [Shr+06], the authors present a HyTM that is fundamentally controlled by software where the hardware is simply used to optimize the performance of transactions. This system embodies this software-centric hybrid strategy which comprises a new coherence protocol proposed by them, which is called Transactional MESI (TMESI) as well as an STM system.

2.2.4 Software transactional memory systems

Usually, STM systems can be either implemented in a library or directly into a compiler. The library-based solution requires the programmer to add explicit calls to the STM library for every access to the memory inside a transaction. This approach is applicable in every system but requires significant changes to the application source code when compared to the sequential code. In order to address this issue, a compiler or preprocessor can be used to automatically convert all accesses to shared memory

executed within transactions (*e.g.*, delimited by **atomic blocks**) into proper function calls of an STM library.

The second approach is to use a compiler which includes all STM functionalities. In this case, the compiler includes the support to STM for a specific programming language. The scope of software support includes language extensions to specify and define transaction regions (atomic blocks).

There exist a large variety of programming languages that can make use of STM (Figure 2.8). Some languages have STM support implemented in their compilers. This is the case of the GNU Compiler Collection (GCC) 4.7 [GCC12] (C/C++), the Intel C++ STM Compiler Prototype Edition [Int09] (C++) and the Glasgow Haskell Compiler [GHC12] (Haskell). In other cases, STM is available as a library without having any specific integration with the compiler.

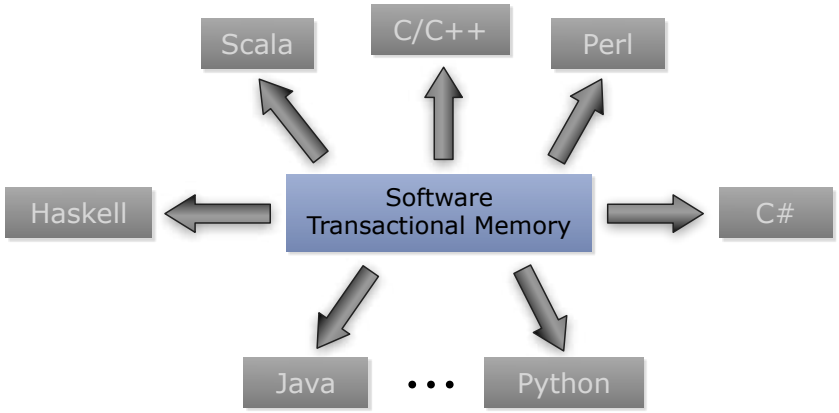


Figure 2.8: STM integration with programming languages.

In this thesis, we are interested in STM systems implemented in C/C++ as software libraries as well as TM applications that explicitly call STM functions in the source code. Implementations such as TL2 [DSS06], TinySTM [FFR08], SwissTM [DGK09] and RSTM [Mar+06] use the library approach. The analysis of the influences of preprocessors that convert atomic blocks into STM system calls was done in [Fel+07] and is out of the scope of this thesis.

TL2

The Transactional Locking II (TL2) algorithm is the second version of the original Transactional Locking (TL) algorithm developed by *D. Dice* and *N. Shavit* [DS06]. Based on a global versioning approach, and in contrast with prior local versioning approaches, the authors were able to eliminate several key safety issues afflicting other lock-based STM systems and simplify the process of mechanical code transformation [DSS06].

The basic idea of TL2 algorithm is to use a *global version-clock* counter in order to handle conflicts between transactions. The counter is incremented using an *increment-and-fetch* function implemented with a *compare-and-swap* (CAS) operation. This global variable will be read and incremented by each *writing transaction* and will just be read by every read-only transaction.

A similar idea of the global version-clock variable is used to implement a data structure responsible for storing a collection of special *versioned write-locks* used for every memory location accessed inside transactions. In this case, a single bit is used to indicate whether the lock is taken. The rest of the lock word is used to store a version number. This version number is incremented by every successful release of the respective lock. It is also possible to associate locks and shared data in different ways such as *per object* (a lock is assigned per shared object) or *per stripe* (the shared memory is partitioned using some hash function to map a striped location to a lock).

The sequence of operations that are performed by the TL2 when a transaction begins depends on the type of the transaction. One of the goals of the TL2 algorithm is to offer an efficient execution of *read-only transactions*. In this case, few steps are executed allowing low-cost read-only transactions. On the other hand, *write transactions* (*i.e.*, transactions that performs writes to the shared memory) need more steps and may have a significant cost depending on the operations that are executed inside them.

TinySTM

TinySTM is another well-known STM implementation that also uses a global versioning approach (shared counter as clock) to control the conflicts between transactions and locks to protect shared memory locations [FFR08]. It also uses a

shared array of locks to manage concurrent accesses to memory. Each lock covers a portion of the address space and the addresses are mapped to locks based on a hash function. Since write-transactions must verify that all the addresses they have read are still valid at commit-time (*i.e.*, they are not locked by another transaction and still have the same version number), depending on the number of read and write operations, this verification may be costly. In order to address this issue, the authors propose a hierarchical locking strategy. In this strategy, the leaves of the tree (last level on the hierarchy) correspond to elements of the shared array of locks while upper levels aggregate information about lower levels. Thus, when there is no lock acquired for any element of a given sub-tree, is not necessary to validate its elements.

TinySTM implements different designs that mainly differ in the moment that locks are acquired (*encounter-time* or *commit-time*) and in the way updates are written to memory (*write-through* and *write-back*). With encounter-time locking, locks are acquired when the transaction encounters write operations. Differently, commit-time locking delays the acquisition of locks to the commit phase. With write-through access, updates are written directly to memory and old values are stored in an undo log to be restored upon abort. With write-back access, updates are buffered during the transaction execution. Then, they are written to the memory upon commit.

Each strategy has its advantages and limitations. Write-through has lower commit-time overhead, faster read-after-write/write-after-write handling and enables many interesting compiler optimizations. On the other hand, write-back has lower abort overhead and does not require extra techniques to guarantee consistent reads. These configurations can be set during the compilation, but TinySTM uses write-back with encounter-time locking as its default configuration. It is also possible to use different contention management strategies implemented in TinySTM. The following are the most used ones:

- **suicide**: immediately aborts the transaction that detects the conflict (this is the default strategy);
- **delay**: is similar to suicide but waits until the lock that caused the abort has been released before aborting the transaction;
- **backoff**: is similar to suicide but waits for a random delay before aborting the transaction (the delay increases exponentially with every abort).

SwissTM

SwissTM [DGK09] is a very recent STM implementation which has some similar characteristics when compared to TL2 and TinySTM. It is a lock-based STM, which means that it uses a lock table to manage concurrent accesses to memory. SwissTM is also word-based as it enables transactional access to arbitrary memory words (word granularity).

However, SwissTM presents some new features when compared to TL2 and TinySTM. One of its innovations is the hybrid conflict detection scheme: it detects write/write conflicts eagerly, which prevents transactions that will probably abort from running and wasting resources, and read/write conflicts lazily, allowing more parallelism between transactions. In read/write conflicts, a time-based scheme (similar to the TL2 global version-clock) is applied to handle conflicts.

Another distinctive feature of SwissTM is its two-phase contention manager. Short or read-only transactions use the simple but inexpensive *timid contention management* scheme, aborting transactions when the first conflict is encountered. On the other hand, more complex transactions are switched dynamically to a mechanism that involves more overhead but favors theses transactions, preventing starvation. More precisely, the mechanism that is applied for such complex transactions is called Greedy [GHP05].

RSTM

Rochester Software Transactional Memory (RSTM) is one of the oldest open-source Software Transactional Memory systems [Mar+06]. Differently to the previous discussed STM systems, RSTM is a C++ library for object-oriented transactional programming. Since its first release in 2006, it has grown to include several STM algorithms, allowing the system to be customized to a given workload. All algorithms are blocking and all operate at the granularity of individual words. It also supports several architectures and operating systems such as x86 / SPARC, Linux, Solaris and MacOS.

Since RSTM implements several STM algorithms, it can be configured to behave similar to other STM systems such as TL2, TinySTM and SwissTM. This means that RSTM can use the core algorithms of other STM systems but their implementations

can differ on several aspects: how logging is performed, the write set implementation for redo-log algorithms and how memory management is done. One optimization implemented in RSTM concerns the cache accesses. RSTM reduces cache misses by employing one single level of indirection to access shared objects. It means that each object has a unique metadata structure during its lifetime, avoiding the creation of a new locator whenever a object is acquired by a transaction.

2.3 Benchmarks for evaluating transactional memory systems

Since the appearance of the first TM proposal, we have seen several efforts to develop TM microbenchmarks and benchmarks to evaluate TM implementations. In this section, we discuss the most known microbenchmarks, realistic benchmarks suites and useful tools to generate synthetic TM applications.

2.3.1 Data structure-based microbenchmarks

These microbenchmarks are based on single data structures to evaluate TM implementations. They are useful for constructing basic-level insights of TM designs and can be parameterized to conform with the user needs. Examples of common parameters are the percentage of insertions, deletions and lookups inside the data structure. However, they do not exhibit a wide range of TM characteristics [Kes+09] and do not represent realistic workloads.

As examples of such data structure-based microbenchmarks we can cite the red-black tree [DSS06], hash table [Dic+09], linked list [FFR08], skip list [Dra+11] and rand-array [Dam+06].

2.3.2 Realistic benchmarks

Since microbenchmarks are limited their simplicity, it emerged the necessity of having realistic benchmarks to evaluate TM implementations in real-life workloads. In this section, we discuss the most known realistic benchmarks to evaluate TM.

STAMP benchmark suite

Stanford Transactional Applications for Multi-Processing (STAMP) [Min+08] is a benchmark which includes 8 applications and 30 variants of input parameters and data sets. The set of applications consists of a variety of algorithms and different application domains. These applications are composed of a wide range of transactional behaviors such as the size of transactions, amount of contention, sizes of read and write sets, coarse-grain and fine-grain transactions. Additionally, it offers a good portability, since it is possible to run many classes of TM systems, including HTM, STM and HyTM designs. The summary of the STAMP applications is presented below:

- **bayes** implements an algorithm for learning the structure of Bayesian networks from observed data. It uses different data structures such as adtrees and a directed acyclic graph. On each iteration, each thread is given a variable to analyze and as more dependencies are added to the network, connect subgraphs of dependent variables are formed. Transactions are used to protect the calculation and inclusion of new dependencies in the graph. This application is characterized by its high contention and most of its execution time is spent inside long transactions.

- **ssca2** is composed by four graph kernels that operate on a large, directed, weighted multi-graph. However, only Kernel 1 is well suited for TM and was then parallelized. This kernel constructs an efficient graph data structure using adjacency arrays and auxiliary arrays. The parallel transactional version of such kernel is composed of threads that add nodes to the graph in parallel. In this context, transactions are used to protect accesses to the adjacency arrays. The main characteristics of the parallel transactional implementation are: low time spent in transactions, small size of transactions and read and write sets and relatively low contention (infrequent concurrent updates of the same adjacency list).

- **genome** takes a large number of DNA segments as its input parameter and tries to mach them to reconstruct the original source genome. This process is composed by two phases. The first phase of the algorithm uses a hash set to

create a set of unique segments, excluding all duplicates. After that, the second phase is executed by many threads where each one tries to remove a segment from a global pool of unmatched segments and add it to its partition of currently matched segments. Additions to the set of unique segment and accesses to the global pool of unmatched segments are enclosed by transactions to allow concurrent accesses. The main characteristics of the parallel transactional implementation are: transactions with moderate length, moderate read and write set sizes and almost all of the execution time is transactional with low contention.

- **intruder** emulates Design 5 of the Signature-based network intrusion detection system, which scans network packets in order to detect a known set of intrusion signatures. It is composed by three phases: capture, reassembly and detection. Different shared data structures are used depending on the phase: a FIFO queue is used in capture whereas a dictionary implemented by a self-balancing tree is used in reassembly phase. Both capture and reassembly phases are enclosed by transactions. This application is composed of short transactions but presents high contention.

- **kmeans** is a clustering algorithm that tries to group similar elements into K clusters. It iterates over a set of elements and calculates the distance between these elements and their centroids. Transactions are used to protect the update of the cluster center that occurs during each iteration. Since threads only occasionally update the same centroid concurrently, this algorithm is well suited for TM. It has short transactions and presents low contention.

- **labyrinth** is a variant of Lee's routing algorithm [JK81] implemented with transactions. The calculation of the path is enclosed by a single transaction and a conflict occurs when two or more threads pick paths that overlap. Transactions are beneficial for implementing this solution since deadlock avoidance techniques are required when implementing it with locks. It has a very high number of short transactions (few instructions inside transactions) as its main characteristic.

- **vacation** emulates an on-line travel reservation system. Each client has a fixed number of requests generated randomly in a distributed fashion. Each request is enclosed in a single transaction that performs the accesses to the database server. The system keeps track of customer reservations through a set of shared trees. This application spends a lot of time inside transactions and has medium length transactions.

- **yada** implements Ruppert's algorithm for Delaunay mesh refinement [KCP06]. It consists of a shared graph structure (mesh) where each node is a triangle, a set of segments that delimits the mesh boundary and a shared queue of bad triangles (*i.e.*, triangles that do not satisfy a quality criteria). The refinement is an iterative process. In each iteration, a bad triangle is removed from the queue, its retriangulation is performed on the mesh and new bad triangles that result from the retriangulation are added to the queue. The computation finishes when there is no more bad triangles in queue. Transactions protect the queue as well as the entire refinement of a bad triangle. It has long transactions and most of its execution time is spent in transactions.

Lee-TM

Lee-TM [Ans+08] is another implementation of the Lee's routing algorithm used in circuit routing [JK81]. Differently from the STAMP's labyrinth, Lee-TM also provides different lock-based (coarse-grain and medium-grain) and transactional (non-optimized and optimized) implementations. This enables direct performance comparisons between lock-based and transactional versions.

RMS-TM

The RMS-TM benchmark [Kes+09] is composed of applications from the Recognition, Mining and Synthesis (RMS) domain. More precisely, this benchmark includes some RMS applications from the benchmarks BioBench [Alb+05] and MineBench [Nar+06] that can benefit from using TM. The applications can be executed with different data set sizes. Differently from the previous presented benchmarks, it also includes I/O operations inside transactions.

To transactify the selected applications, the authors replaced locks used to protect the accesses to shared variables with transactions using a prototype of the Intel C/C++ compiler with STM support [Int09]. As we previously mentioned in this chapter, the Intel STM compiler includes language extensions to specify and define transaction regions. The summary of the RSM-TM applications is presented below. *Hmmsearch* is from BioBench whereas the others are from MineBench.

- **Hmmsearch** searches for homologous protein or nucleotide sequences. More precisely, this application reads a Hidden Markov Model (HMM) and searches a sequence database for significantly similar sequence matches. In the transactional version, threads read from an input list of sequences in parallel and use transactions to protect the accesses to this input list. Additionally, transactions are used to protect the accesses to two score lists and a histogram of the whole sequence. This application presents a very low abort ratio and short transactions.

- **Apriori** is a iterative mining algorithm for learning association rules. In each iteration, only the itemsets that meet a minimum support criterion in the previous iteration generate a new candidate set. Then, it checks whether all the subsets of the itemset have a required support value before inserting an itemset into a new candidate set. It uses a breadth-first search and a hash tree structure to count candidate item sets efficiently. Transactions are used to protect the calculation of support values as well as the accesses to the hash. This application has short transactions and a high abort ratio.

- **Utility-Mine** is another mining algorithm for learning association rules. However, it tries to identify itemsets with high *utilities*. The utility of an itemset is defined as how useful the itemset is. It consists of two phases: it first generates candidate itemsets on the search space and then defines the high utility itemsets by scanning the transaction database. Transactions protect the update of the utility of itemsets and insertions of candidates into the shared hash tree. This application has long transactions and a very low abort ratio.

- **ScalParcC** is a parallel formulation of a decision tree classification. The goal of this algorithm is to create a model that predicts the value of a target variable

based on several input variables. To do so, the algorithm uses a hash table as its main data structure. Transactions protect the accesses to the hash as well as some other shared counters. This application has short transactions and medium abort ratio.

2.3.3 Highly configurable workload generators

Another set of benchmarks aim at exploring a wide range of transactional behaviors or stress particular aspects of TM. We call these benchmarks as "high configurable workload generators", since they usually several configuration parameters that can be easily changed to produce workloads for evaluating TM systems. They also have the ability to mimic the behavior of real TM applications. In the following sections we describe the most used ones.

EigenBench

EigenBench [Hon+10] is a lightweight microbenchmark that allows users to perform a thorough exploitation of the orthogonal space of TM applications characteristics. Due to its several parameters, it can be easily configured to mimic a wide variety of workloads.

The core of EigenBench is very simple: each thread performs a pre-defined number of transactions then exits (Figure 2.9). A transaction consists of a set number of transactional read and write operations (Figure 2.9, lines 12-18), with some non-transactional (*i.e.*, local) operations inbetween (Figure 2.9, line 20). Additional local operations can be performed between transactions (Figure 2.9, line 24). Those operations are performed on three separate array structures during the execution. The *hot array* (`A1`) is shared between all concurrent threads and it is accessed transactionally. The *mild array* (`A2`) is also accessed transactionally but each thread accesses its own partition (*i.e.*, there is no conflicts). Finally, the *cold array* is partitioned like the *mild array* but it is used for non-transactional accesses. The use of those distinct arrays allows the user to control the contention between the transactions as well as the amount of influence of the non-transactional code on the workload.

```
1  void test_core(tid, loops, pesist, lct, R1, W1, R2, W2
2      R3_i, W3_i, Nop_i, k_i, R3_o, W3_o, Nop_o, k_o) {
3    long val=0;
4    long total = W1 + W2 + R1 + R2;
5    for (i=0; i<loops; i++) {
6      Save_Random_Seed;
7      BEGIN_TM();
8      if (persist) Restore_Random_Seed;
9      (r1,r2,w1,w2) = (R1,R2,W1,W2);
10     Reset_History_Buffers;
11
12     for (j=0; j<total ; j++) {
13       (action, array) = rand_action(r1, w2, r2, w2);
14       index = rand_index(tid, lct, array);
15       if (action == READ)
16           val += TM_READ(array[index]);
17       else
18           TM_WRITE(array[index], val);
19       if ((j%k_i)==0)
20          val += local_ops(R3_i, W3_i, Nop_i, val, tid);
21     }
22     END_TM();
23     if ((i%k_o)==0)
24       val += local_ops(R3_o, W3_o, Nop_o, val, tid);
25 } }
```

Figure 2.9: Pseudo-code description of EigenBench extracted from [Hon+10].

The `rand_action()` function is used to decide from a read or write operation and whether to access the *hot array* or *mild array*. It also guarantees a precise number of reads and writes to each array, according to the specified parameter values. However, the order in which they are accessed is randomized. The `local_ops()` performs a given number of read or write operations on the *cold array*. The number of local operations can be either more (if `k_i` and/or `k_o` are equal to 1) or less (if `k_i` and/or `k_o` are greater than 1) frequent than shared operations. The `rand_index()` is responsible for determining the index of the selected array that must be accessed. This function can also be parametrized to use random indexes or to control the probability of using previously accessed indexes to simulate temporal locality.

Due to the wide range of parameters, this benchmark allows users to create several different workloads. Additionally, it is possible to configure EigenBench to mimic the behavior of real TM applications as shown in [Hon+10].

STMBench7

STMBench7 [GKV07] is an adaptation of the OO7 benchmark [CDN93] implemented with transactions. The OO7 benchmark has been originally designed to compare object-oriented database systems. Although it is not specific to any particular application, it represents a wide variety of commercial applications including Computer-Aided Design (CAD) and Computer-Aided Manufacturing (CAM) systems.

Like OO7, this benchmark operates over a rich object-graph with millions of objects and many interconnections among them. The graph consists of several modules, each containing a tree of assemblies. The benchmark uses more than 40 distinct operations over the graph, each one having a different scope and complexity. Because of that, it can simulates many different real-world scenarios. However, STMBench7 differs from OO7 in two important aspects: (i) it considers various concurrency patterns and workloads whereas OO7 was developed to evaluate the performance of isolated transactions; and (ii) the graph in STMBench7 is highly dynamic.

Overall, the operations performed by STMBench7 can be divided into four main categories: long traversals (go through all assemblies), short traversals (traverse the structure via a randomly chosen path), short operations (choose few objects and perform an operation on the objects) and structure modification operations (create or delete elements). The user describes a target workload by providing the workload type (read-dominated, read-write or write-dominated), the types of allowed operations (*i.e.*, whether long traversals and/or structure modification operations are enabled) and the number of threads.

The STMBench7 is implemented in Java and C++ and also includes implementations with coarse- and fine-grain locks. This allows the users to compare the results obtained with a specific TM implementation against locks.

WormBench

WormBench [Zyu+08] is a configurable TM workload written in C#. The goal of this benchmark is to help TM researchers easily create transactional workloads that can be used for evaluating and verifying the correctness of TM systems.

The idea is inspired from the Snake game. The application has two main data

structures: the *BenchWorld* and the *Worm*. During the execution of the application, Worms move in the BenchWorld and execute some operations from an user specified stream. Worms are active objects meaning that every Worm object is associated with one thread. They can move in different speeds and they can have different sizes. The size of the BenchWorld can also be specified by the user. Since the BenchWorld is shared, depending on its size and the movements and attributes of the Worms conflicts can happen less or more frequently. For instance, reducing the size of the BenchWorld and increasing the number for Worms will also increase the probability of conflicts. Since all these parameters can be chosen, it is possible to create several different workloads.

2.4 Concluding remarks

In this chapter, we reviewed the basic concepts and topics that are relevant to the context of this thesis. First, we discussed that multicore platforms are the mainstream approach to deliver high performance. However, achieving the maximum possible performance from those platforms is a daunting task. This is due to the fact that each multicore platform has its peculiarities such as the way computing units are distributed throughout the platform and the memory hierarchy. In addition to that, synchronizing accesses to shared data is challenging and can limit the scalability of parallel applications.

Second, we showed that Transactional Memory appears as an alternative synchronization mechanism adapted for multicores. In contrast with the low-level mechanisms such as locks and semaphores, the Transactional Memory programming model offers a higher abstraction level to deal with concurrency. It shifts the problem of correct synchronization to the Transactional Memory system, which is responsible for making sure that deadlocks will not occur and locks are performed at a granularity which allows to indeed exploit the inherent parallelism of the applications. We also discussed that different implementations of Transactional Memory systems make dissimilar design choices, which can have an impact on the performance of applications. More precisely, in this thesis we are interested in Software Transactional Memory because hardware support is still in the premature stage.

Finally, we gave a brief overview of some of the most used benchmarks for

evaluating Transactional Memory systems. Since there is still a limited number of real-world applications that make use of Transactional Memory, these benchmarks are essential to implement, test and evolve Transactional Memory systems.

The most used benchmark suite in the literature is STAMP. We believe that this is due to the fact that it has several real-world applications from different domains and it is also highly configurable. We agree with that point and we also decided to use STAMP throughout this thesis due to the same reasons. Among the highly configurable workload generators, we believe that EigenBench is the most interesting one due to its simplicity and its potential to explore several different parameters that are relevant in the context of TM. This is the reason why we also use EigenBench in this thesis.

CHAPTER 3

Understanding the Performance of TM Applications

ALTHOUGH TM promises to substantially simplify the development of correct concurrent programs, programmers still need to debug code and study ways to optimize TM applications. In this chapter, we discuss some important topics concerning the performance of TM applications. We begin by showing that it is not trivial to understand the performance of TM applications. We demonstrate that, depending on the characteristics of the parallel application, the use of STM may present in some cases similar performance than traditional synchronization mechanisms or, in other cases, it can incur in higher overhead (Section 3.1).

In Section 3.2, we show that the performances of applications using TM-based synchronization also depend on the STM system specifics. In order to gain some insight on these issues, helping developers to understand and improve the performance of TM applications, we propose to trace transactions and we discuss what events are relevant and thus must be collected (Section 3.3). Then, in Section 3.4, we discuss the details about the implementation of the tracing mechanism. We show that our solution can be applied to different STM systems and applications as it does not modify neither the target application nor the STM system source codes. Finally, we discuss how our approach can be used to analyze and better comprehend the performance of TM applications through three case studies (Section 3.5). Finally, we

conclude this chapter in Section 3.6.

3.1 STM vs. traditional synchronization

The performance and benefits of using TM have been discussed since its first proposal in 1993 [HM93]. This discussion has been strengthened after the appearance of the first STM system. In terms of performance, the research community tends to claim that the overall performance of STM is significantly worse at low levels of parallelism when compared to traditional shared-memory programming [Cas+08]. However, this statement is not always true and it is not easy to predict the performance of a TM application.

Let us consider again the TSP application that we used to demonstrate how locks can limit the scalability in Section 2.1.4. In addition to the two variations of TSP using locks (strict and relaxed), we present a third variation using STM. This variation was obtained from the **lock-strict** variation by replacing all mutex locks that protected the accesses to shared variables by transactions. Figure 3.1 shows the same portion of the source codes of both lock-strict and STM variations.

```
lock (lock_minimum);
if length ≥ minimum then
    (*cuts)++;
    unlock (lock_minimum);
    return 1;
end
unlock (lock_minimum);
```

(a) lock-strict

```
tm_begin ();
if length ≥ tm_read (&minimum) then
    (*cuts)++;
    tm_end ();
    return 1;
end
tm_end ();
```

(b) STM

Figure 3.1: Snippets of the lock-strict (left) and STM (right) versions of TSP.

Since in this case we did not use a compiler that supports TM syntax, all mutex locks were manually replaced by function calls to the underlying STM system. These functions indicate the beginning (`tm_begin`) and the end (`tm_end`) of a transaction. Additional function calls were inserted to inform the STM system that a shared

variable is read (`tm_read`) or written (`tm_write`) inside a transaction and thus must be consistent in regard to other concurrent transactions.

Figure 3.2 shows the mean execution times of 30 executions of these three approaches for different thread counts. The underlying STM system used for this experiment was TinySTM with its default configuration. In all experiments, we used the same seed for the pseudorandom number generator and a graph composed of 16 nodes.

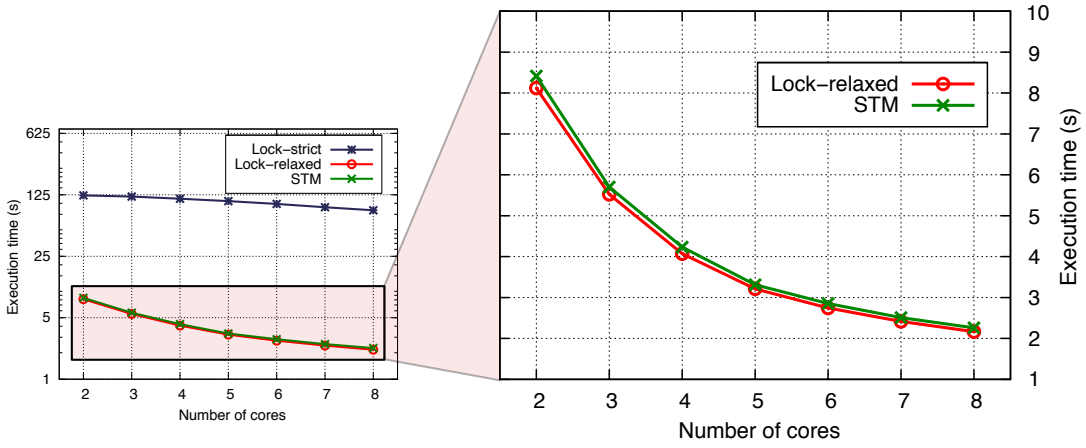


Figure 3.2: Execution times of the three variations of TSP (locks vs. STM).

Although the STM variation is very similar to the lock-strict (in the sense that all shared accesses are protected), the results obtained with the STM variation are very similar to those of the lock-relaxed variation. The STM did not add a considerable overhead. We indeed observed a little overhead with low thread counts but on average it was about 3.6% worse than the lock-relaxed variation. This shows that, depending on the applications characteristics, STM can present comparable performance to locks.

The high contention generated by the lock-strict variation is not present in the STM variation due to two reasons: (i) the STM uses an optimistic approach to handle multiple accesses to shared data, so threads are not blocked most of the time; and (ii) most of the accesses do not conflict, which benefits such optimistic approach.

The TSP is an example of a good candidate for TM, since it can benefit from the TM optimistic approach. However, for more complex applications it may not be trivial to know if they will perform well with TM. Those applications include the

ones that present transactions with higher probability of conflicts or transactions that conflict several times before committing. For those cases, the STM system can also play an important role. In the next section we analyze the impact of STM systems on the performance of more complex TM applications.

3.2 Performance impact of STM systems

As previously mentioned, the performance of a TM application can also be impacted by the underlying STM system. On the STM system side, this impact comes from its design choices, such as the version management and conflict detection, discussed in [Wan+11] and resolution mechanisms, which was recently discussed in [HYH12]. Since STM systems implement different strategies, we expect that TM applications will perform differently depending on the underlying STM system.

In order to investigate such impact, we have carried out experiments with all non-trivial TM applications available from the STAMP benchmark suite. STAMP offers some important advantages over other TM benchmarks: the applications use a variety of algorithms and belong to different application domains and the applications can be easily executed with different STM systems. In order to stress the STM systems, we chose the largest input sizes as described in the STAMP's original paper [Min+08]. To cover a wider range of transactional characteristics, we used the low contention parameters for vacation and kmeans.

We executed the STAMP applications with four state-of-the-art STM systems, namely TinySTM [FFR08] (version 1.0.3), TL2 [DSS06] (version 0.9.6), SwissTM [DGK09] (version 2011-08-15) and RSTM [Mar+06] (version 7). They represent the most known STM systems implemented in C (TinySTM and TL2) and C++ (SwissTM and RSTM) available to date. Since some STM systems can be configured to use different algorithms, we used in all experiments their default configurations. Each experiment was executed at least 30 times and the speedups represent average execution times obtained on the SMP-24 platform (a detailed information about this platform is presented in Section 5.1.1). It is also important to mention that all experiments were carried out with exclusive access to the multicore machine and we used a static thread mapping to avoid thread migrations and possible different thread placements applied by the Linux scheduler. The aforementioned static thread

mapping fixes one thread per core and places threads as close as possible to profit from cache sharing.

Figure 3.3 presents the speedups of all STAMP applications with the four STM systems.[1] As it can be noticed, the performance of the applications may be considerably impacted depending on the STM system used. Overall, TinySTM and SwissTM presented better results than other STM systems.

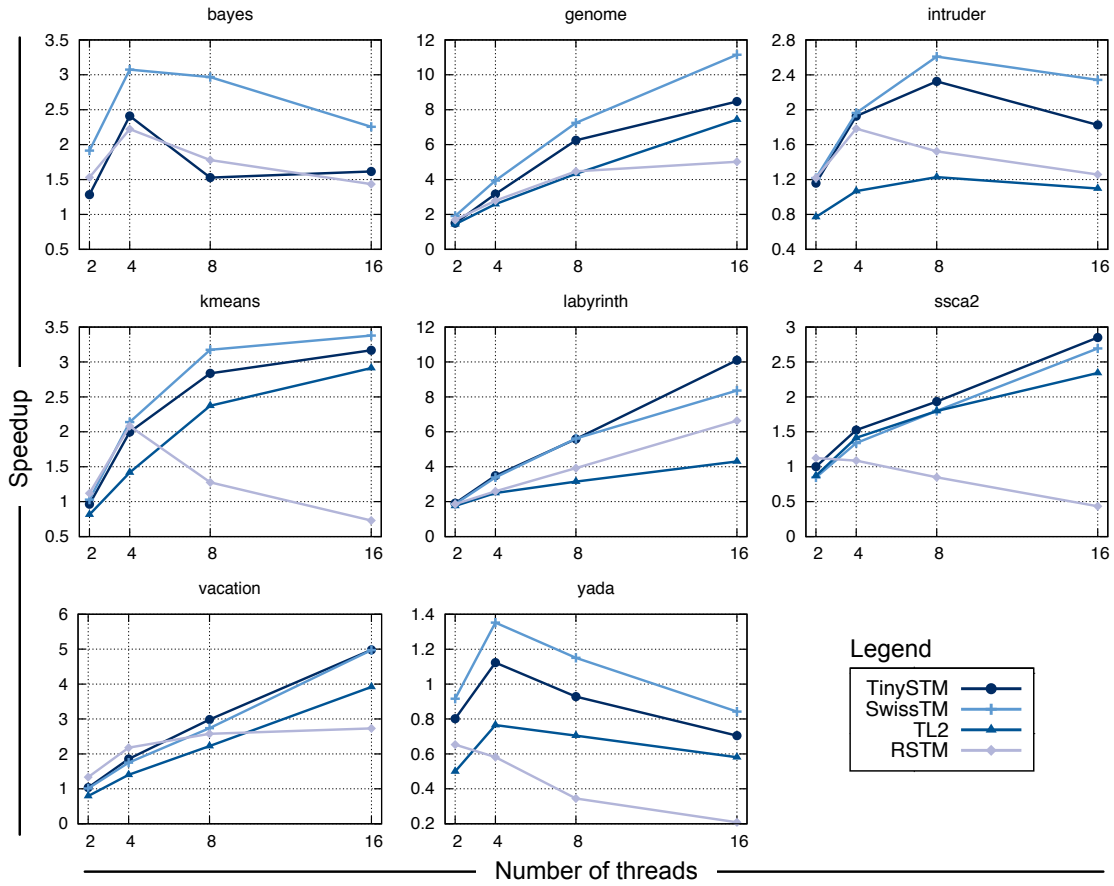


Figure 3.3: Speedups off all STAMP applications with four state-of-the-art STM systems.

Considering the two most performant STM systems (TinySTM and SwissTM), we can draw some important conclusions. The application bayes presented an

[1]We did not include the results of bayes with TL2, since this configuration led to incorrect executions of this application.

unpredictable behavior. Mainly with TinySTM, the execution time varied considerably which makes difficult the performance comparison with SwissTM. Vacation and ssca2 presented very similar performances.

Three interesting cases were intruder, labyrinth and genome. The performances obtained with both STM systems on intruder were fairy the same with low thread counts. However, with 8 and 16 threads, SwissTM performed better. On labyrinth, we noticed a similar behavior considering low thread counts but TinySTM performed better than SwissTM when the number of threads was 16. Finally, SwissTM presented better performance for all thread counts on genome, increasing the gain as the number of threads increases.

Unfortunately, it is not trivial to understand those performance impacts. We believe that we can gain some insight on these issues by using tracing techniques. In the next section, we propose to trace relevant runtime events derived from the use of STM. Then, in Section 3.4, we present a generic approach to collect and trace information from transactions and we discuss some of its implementation details. Finally, in Section 3.5, we use such approach to better understand the performance of those three interesting cases.

3.3 Tracing TM applications

STM systems intend to simplify the programming problems deriving from the use of low-level synchronization mechanisms. However, such simplicity hides from the developers several issues which impact the performance of the applications. Some STM systems maintain some statistics about the number of commits and aborts after the execution of the TM application. Although these statistics can be useful to confirm if the TM application is high- or low-conflicting, they only show a global view of the application behavior. For instance, it is not possible to know if the high number of conflicts occurs during a specific execution phase or if conflicts are spread across the whole execution.

One technique to get such temporal information from events generated by the applications is tracing. Tracing applications basically consists in recording a chronological history of these events, representing the application behavior. An event is an action during the execution of an application that changes its state.

There exist some tools that help developers to collect and trace events during the execution of parallel applications. As examples, we can cite strace, Linux Trace Toolkit next generation (LTTng) [DD06], Tuning and Analysis Utilities (TAU) [SM06] and Performance Application Programming Interface (PAPI) [Ter+10]. The strace tool provides a primitive form of userspace tracing. However, it is limited to tracing system calls and signals, both of which are nowadays obtainable at a much lower cost through kernel tracing. LTTng is a well-known tool for tracing the Linux kernel. Although it provides a highly efficient kernel tracer, it lacks an user-space tracer of equal performance. TAU is a program and performance analysis tool framework. It maintains performance data from parallel programs for each thread, context, and node in use by an application. Finally, PAPI provides an interface that allows users to easily collect hardware performance counters.

The previously cited tools are used in different contexts. The LTTng is useful for tracing events from the kernel whereas TAU is convenient for gathering events from the middleware such as the communication primitives in applications implemented with Message Passing Interface (MPI). PAPI is useful when the user needs information from events generated by the hardware (*e.g.*, number of instructions and cache and memory accesses).

In this thesis, we are specifically interested in events deriving from the use of STM. This allows us to propose an approach to gather useful information with low intrusiveness and independent of the TM application and STM system. In the following sections, we first specify the desired characteristics of our approach. Then, we explain the general functioning of STM systems and what events we intend to trace. Finally, we discuss its implementation details.

3.3.1 Goals

We believe that a tracing mechanism adapted to TM must be generic enough to deal with the high variety of STM systems and TM applications. In order to do so, such mechanism has to tackle the following three requirements:

- **STM system independency**. There exist several distinct STM systems currently available, each one having its peculiarities. We want a tracing solution that

could be easily applied in different STM systems without any modification in their original source codes.

- **TM application independency**. In theory, TM applications can use any STM as their underlying system to implement all TM functionalities. We thus want a tracing solution that could be used on TM applications without any modification in their source codes.

- **Low intrusiveness**. The tracing mechanism should minimize intrusiveness, meaning that it should not imply an important execution overhead on both TM application and STM system. Indeed, when such an overhead is important, the application behaves differently and traces may not represent the real behavior of the application.

3.3.2 Which events to trace?

Another important question that must be addressed is which events are relevant and thus should be traced. Since we target TM applications, we are interested in events derived from the TM. As previously discussed in Section 2.2.2, STM systems differ in many aspects. However, all of them rely in a small set of operations to provide the basic TM functionalities. These operations are implemented as functions in STM systems and they appear in the TM application source code as function calls to the STM system.

Event	Function (TinySTM)	Action performed
StmInit	`stm_init()`	Initialize the STM system.
StmExit	`stm_exit()`	Finalize the STM system.
TxBegin	`stm_start()`	Start a transaction.
TxEnd	`stm_commit()`	Commit the current transaction.
TxAbort	`stm_rollback()`	Restart the current aborted transaction.
TxRead	`stm_load()`	Transaction executes a read operation.
TxWrite	`stm_store()`	Transaction executes a write operation.

Table 3.1: The most common STM operations.

In Table 3.1, we define the most important events that can be generated from the STM system. We also correlate these events with their corresponding functions in TinySTM[2].

Tracing the above TM events allows us to gather useful information from the TM applications. Obviously, the information we need to obtain from the TM application dictates the set of events that we must trace. Additionally, the intrusiveness on the TM application also depends on the traced events.

For instance, if we intend to compute statistical information about the read/write rates for each transaction in the application we register the events $TxRead$ and $TxWrite$. In this case, we may not be interested in registering the timestamp of each event because we do not need temporal information. Instead, we may simply implement counters to store the number of read and write operations for each transaction, incrementing them each time a transaction performs read/write operations.

However, if we intend to gather data to be visualized in time-based behavior information charts, *e.g.*, the commit rate[3] during the execution of the application, we do not need to register the events $TxRead$ and $TxWrite$. Instead, we just register the events $TxEnd$ and $TxAbort$ with their respective timestamps. This allows us to obtain the commit rate behavior along the execution time.

3.4 A tracing mechanism adapted for TM applications

We implemented our tracing solution based on the Linux dynamic linking mechanism which provides a simple way to intercept function calls. Linux provides the environment variable called `LD_PRELOAD` which is used to dynamically load a library LIB when launching applications. During the execution, our tracer intercepts the functions having the same signatures as the ones implemented in LIB, calling the

[2]There is no standard in terms of TM function names (API). Although the name of these functions may differ depending on the STM system, all STM systems usually implement such functionalities.

[3]The commit rate of an interval $[a; b]$ is given by the number of events $TxEnd$ divided by the number of events $TxEnd + TxAbort$ during the interval $[a; b]$.

corresponding *LIB* functions (*wrappers*). Wrapper functions allow users to easily include additional code before and/or after calling the original functions.

We implemented a prototype shared library called `libTraceSTM`, which incapsulates wrappers for the STM functions that should be intercepted. The TM application is then executed along with the tracing mechanism as follows (where `<app>` is the target TM application and `<app parameters>` are the application parameters):

```
LD_PRELOAD=libTraceSTM.so <app> <app parameters>
```

When executing the TM application with `LD_PRELOAD`, the original STM functions are dynamically overridden by the wrapper functions implemented in `libTraceSTM`. These wrappers are responsible for tracing and calling the corresponding original functions. Figure 3.4 illustrates our tracing mechanism. In this example, we traced the following events: $TxEnd$ and $TxAbort$.

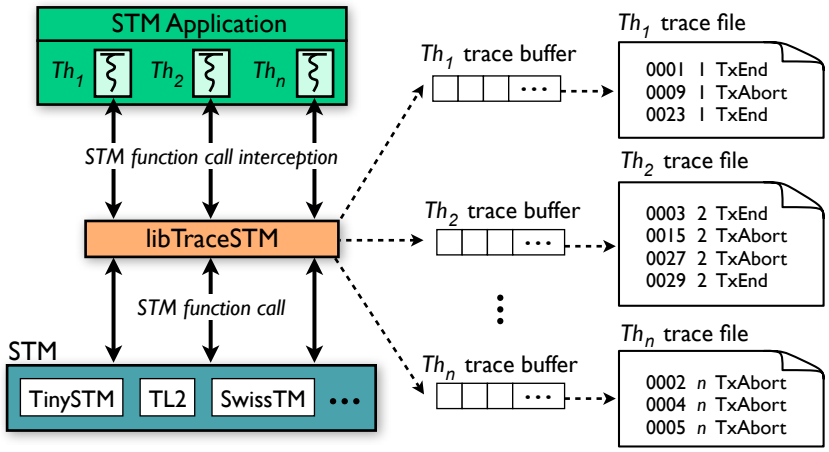


Figure 3.4: Overview of the tracing mechanism.

When a thread is initialized by the TM application, the `libTraceSTM` automatically creates an internal data structure to store information about this thread, creates a trace file and instantiates a memory buffer to temporary store traced events derived from this thread. For the subsequent intercepted STM function calls, the traced events are written into the corresponding thread buffer. When the buffer becomes full, its contents are flushed to the corresponding trace file.

Each traced event has the following attributes:

- **Timestamp**: the time instant in which the event occurred;
- **Thread id**: the identifier of the thread that executed the operation;
- **Event**: the name of the event;
- **Specific**: additional user-defined information (if needed).

Since we obtain one trace file per thread, individual trace files may be merged into a single file following a global order of events. We thus implemented a tool that performs a merge sort of the individual trace files considering their timestamps. The format of the merged trace file is the same of individual trace files. Figure 3.5 shows the result of the merge sort considering individual trace files.

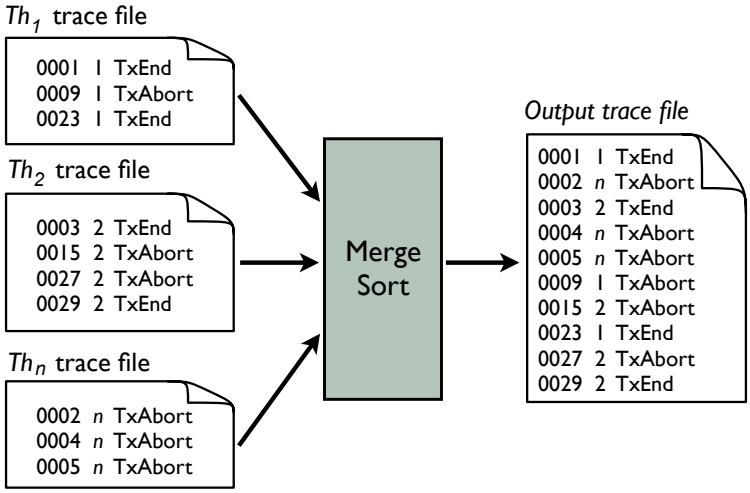


Figure 3.5: Merge sort of individual trace files.

We believe that our approach to intercept STM function calls is very simple and it can be easily extended if more functions should be traced. It is also generic enough, since it can be used with all STM systems and it does not change neither the TM application nor the STM system source codes.

Our prototype tracer is implemented in C and requires any Unix-like system with the GNU Compiler Collection (GCC) installed. It targets STM systems implemented either in C or C++. A mounted temporary file storage facility (tmpfs) for storing temporary individual trace files is highly recommended to reduce the intrusiveness but it is not a hard requirement.

In order to be able to intercept the STM functions, the STM system must be compiled as a dynamic library. By default, most of the state-of-the-art STM systems are compiled as static libraries. However, compiling them as dynamic libraries is straightforward. Once the target STM system is compiled as a dynamic library, TM applications can use such dynamic library instead of the static version without any source code modification.

3.4.1 Function interceptions

Figure 3.6 shows a file diagram of the libTraceSTM main source files. The library is composed of a tracer, files implementing the function wrappers for specific STM systems and a library to manage timestamps. The tracer includes functions to initialize the tracer in the current thread (`tracer_init_thread`) as well as to add events to the trace buffer (`tracer_add_event`) and flush the contents of the buffer into the trace file (`tracer_flush_buffer`). The most important source file is the one that describes the wrappers for a specific STM system. In this example, we show two wrapper files, one for TinySTM and another one for SwissTM to intercept the most common events, *i.e.*, $StmInit$, $StmExit$, $TxEnd$ and $TxAbort$. In addition to these two, we also provide wrappers for TL2. Including function wrappers for other STM systems is straightforward.

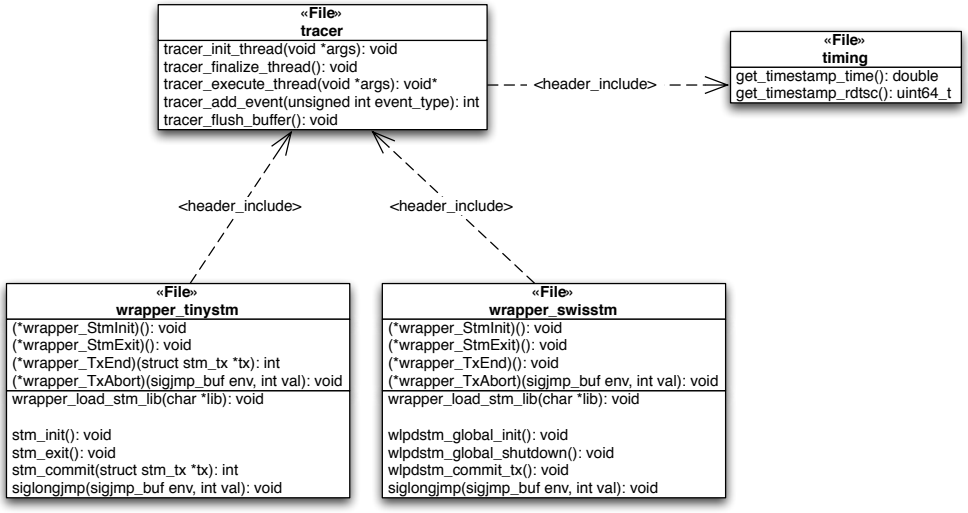


Figure 3.6: File diagram of the main source files of libTraceSTM.

Users can add or remove function wrappers accordingly to their needs. However, there is a set of functions that are intercepted by our tracing mechanism by default and cannot be removed. The first function is `pthread_create`, which is the well know function from POSIX threads to create a thread. We intercept this function to perform some actions in the following order. Firstly, we initialize the tracer internal structure of the this new thread. Secondly, we execute the real function that would be executed by this new thread. Finally, we flush the remaining events stored in the temporary trace buffer of every thread when threads are about to finish.

```
void wrapper_load_stm(char *lib) {
   global_handle_pthread = dlopen ("/lib/x86_64-linux-gnu/libpthread.so.0", RTLD_LAZY);
   global_handle_stm = dlopen (lib, RTLD_LAZY);

   wrapper_pthread_create  = dlsym(global_handle_pthread, "pthread_create");
   wrapper_StmInit  = dlsym(global_handle_stm, "stm_init");
   wrapper_StmExit = dlsym(global_handle_stm, "stm_exit");
   wrapper_TxEnd = dlsym(global_handle_stm, "stm_commit");
   wrapper_TxAbort = dlsym(global_handle_stm, "siglongjmp");
}
```

(a)

```
void stm_init() {
   ...
   wrapper_load_stm(tinystm_string_path);
   tracer_init_thread(NULL);
   tracer_add_event(EVENT_STM_INIT);
   (*wrapper_StmInit)();
}
```

(b)

```
void stm_exit() {
   tracer_add_event(EVENT_STM_EXIT);
   tracer_finalize_thread();
   (*wrapper_StmExit)();
}
```

(c)

Figure 3.7: Snippets of the function wrappers to register the events $StmInit$ and $StmExit$ for TinySTM.

The two remaining functions are those responsible for initializing and finalizing the STM system. Let us consider again TinySTM as an example. In TinySTM, these functions are `stm_init` and `stm_exit`. We do the following actions in addition to their original code in TinySTM (Figure 3.7b and Figure 3.7c). The function `stm_init` is called by the main thread before executing any transactional code. We intercept this function to load the pthread and the STM dynamic libraries and function wrappers (Figure 3.7a, `wrapper_load_stm`), initialize the tracer's internal structure of the main thread (`tracer_init_thread`) and register the $StmInit$ event (`tracer_add_event`).

Analogously, the function `stm_exit` is called by the main thread before finishing the application execution. We intercept this function to register the $StmExit$ event and flush the remaining events stored in the temporary trace buffer of the main thread as well as to copy the contents of all individual trace files to the disk (`tracer_finalize_thread`).

In theory, all functions implemented inside the STM system can be intercepted by our tracing mechanism. This is due to the fact that most of them are available in the STM interface to be called from outside, *i.e.*, in the TM application source code. The only exception may be $TxAbort$: in some STM systems, the function that performs the rollback operation in case of abort may be implemented in such a way that it cannot be directly called from outside of the STM system (for instance, in C, this function can be declared as `static`). This stems from the fact that this is a function that is used internally by the STM system to restart aborted transactions. However, some STM systems such as TinySTM "exteriorize" this function, for instance by including a function `stm_abort` in the STM interface. In these cases, the function `stm_abort` calls `stm_rollback` but the same problem still remains: aborted transactions derived from the conflict detection are restarted by the STM system, which internally calls `stm_rollback` (not `stm_abort`).

Although this privatization issue, it is still possible to know when transactions are restarted by the STM system. For instance, consider the set of functions implement in TinySTM. The function `stm_rollback` is called when a conflict occurs to restart the transaction. The system restarts the transaction by restoring the environment that has been previously saved by `stm_start`. To manage contexts/environments, both functions use the following system calls: `sigsetjmp` and `siglongjmp`. The system call `sigsetjmp` used inside `stm_start` saves the stack context/environment of the calling thread for later use. On the other hand, the system call `siglongjmp` restores the previously save stack context/environment and is called inside `stm_rollback` when the transaction is about to abort. The majority of STM systems implemented in C/C++ use such mechanism to restart aborted transactions. This means that we can capture aborts by intercepting calls to the function `siglongjmp`. This was the strategy we used for TinySTM (Figure 3.7a).

Figure 3.8 shows the wrappers for TinySTM to register the events $TxEnd$ and $TxAbort$. As it can be noticed, the event $TxEnd$ is registered *after* calling the

real function `stm_commit` (Figure 3.8a) whereas the event $TxAbort$ is registered *before* calling the real function `siglongjmp` (Figure 3.8b). This is necessary because when `stm_commit` returns, it indicates that the transaction has committed successfully. In case of aborts, `stm_commit` does not return, calling an internal function (`stm_rollback`) responsible for restarting the transaction. Then, the function `stm_rollback` calls `siglongjmp` at the very end, which restores the context/environment of the calling thread to the beginning of the last executed transaction.

```
int stm_commit(struct stm_tx *tx) {
   int retval;
   retval = (*wrapper_TxEnd)(tx);
   tracer_add_event(EVENT_TX_END);
   return retval;
}
```
(a)

```
void siglongjmp(sigjmp_buf env, int val) {
   tracer_add_event(EVENT_TX_ABORT);
   (*wrapper_TxAbort)(env, val);
}
```
(b)

Figure 3.8: Snippets of the function wrappers to register the events $TxEnd$ (left) and $TxAbort$ (right) for TinySTM.

In these simple examples we showed how users can intercept STM system functions to register STM events. Since wrapper functions are implemented by the users, they can include any additional code to those functions when necessary. For instance, consider that the user only wants to gather statistical information instead of time-based behavior information. In this case, the user may replace the tracer calls to register events (*i.e.*, function calls to the `tracer_add_event` function) for counters, which will store the number of occurrences of each event.

3.4.2 Timestamps

We provide two methods to manage timestamps. The first one uses the function `gettimeofday`, which is accurate to microseconds and incurs some overhead. Alternatively, timestamps can be represented by the number of clock cycles since the last system reset. Our second solution is to use the Time Stamp Counter (TSC), currently available on all x86 processors and incurs a negligible overhead. The value given by this register can be used to impose a global order to the events and can be accessed by each thread individually without any synchronization. However, the value of the TSC register in each processor may slightly drift from the others. This clock drifting

causes the global ordering of the events to be error prone and thus the merged file is not 100% accurate. By default, we use the TSC register to manage timestamps in our tracer.

3.4.3 Intrusiveness

One of the most important aspects of our tracing mechanism is its low intrusiveness. If the overhead is important, the application may behave differently and the traced data may not represent the real behavior of the application.

We measured the intrusiveness of our tracing mechanism considering two important metrics: the execution time and the number of aborts. If execution times and number of aborts obtained by executing the TM applications with the tracing mechanism are very distinct in comparison to the original ones, we can conclude that our trace mechanism is very intrusive and it may change significantly the behavior of the applications.

Table 3.2 shows such information, comparing these two metrics after executing all applications with and without the tracing mechanism using SwissTM as the underlying STM system. The execution times and the number of aborts were obtained from executions of all applications with SwissTM on the SMP-24. We used 16 threads for all runs and we pinned threads to cores to avoid thread migrations. During the execution, we traced the following events: $StmInit$, $StmExit$, $TxBegin$, $TxEnd$ and $TxAbort$. Since the execution time and the number of aborts can vary from one execution to another, we report average values obtained from at least 30 executions of each experiment.

Overall, our tracing mechanism increased the execution time of all applications by a little factor. Concerning the number of aborts, we can notice that the tracing mechanism did not modify the overall behavior of the applications: the number of aborts remains very close to the one obtained without the tracing mechanism. The number of aborts was slightly reduced on all applications with the tracing mechanism. Since we intercept $TxAbort$ events, there is an overhead added by the tracer every time a transaction aborts, so aborted transactions do not restart immediately after detecting a conflict. This may act as a “contention manager” with a fixed backoff and thus can decrease the number of aborts on applications that have medium or high

number of conflicts.

Application	Without libTraceSTM		With libTraceSTM	
	time (s)	*#aborts*	*time (s)*	*#aborts*
bayes	2.89	268	2.94	260
genome	1.33	19,831	1.41	19,722
intruder	16.39	14,401,540	16.49	14,151,069
kmeans	5.96	4,576,947	6.06	4,551,947
labyrinth	8.56	202	8.65	198
ssca2	7.01	12,305	7.20	12,284
vacation	5.04	2,491	5.09	2,434
yada	15.13	1,256,686	15.18	1,254,240

Table 3.2: Intrusiveness of the tracing mechanism on all STAMP applications.

Among all STAMP applications, genome was the most affected one in terms of execution time (it ran approximately 5.6% slower than without the tracing mechanism). This is mainly due to the fact that genome has a very short execution time (approximately 1 second on the NUMA-24 platform with 16 threads). For all other applications, we observed less than 3% of intrusiveness for both metrics. We performed the same measurements using TinySTM, TL2 and RSTM. The results obtained were very similar to those presented in Table 3.2.

Including the events $TxRead$ and $TxWrite$ considerably increased the intrusiveness on most of the STAMP applications. In some cases, the intrusiveness was increased up to approximately 12%. The reason for intercepting $TxRead$ and $TxWrite$ events would be, for example, to detect on which data transactions conflict the most. To reduce the intrusiveness, some modifications on the libTraceSTM can be done to intercept those events only in certain periods during the execution. For instance, one may only intercept those events for transactions executed by a specific application function, where the user wants to obtain more detailed information.

We succeeded on developing a low-intrusive mechanism due to some key points. First of all, our mechanism does not add any synchronization between threads: each thread keeps its traces in a private buffer. Secondly, events are stored in a compact binary format to reduce memory usage[4].

Finally, our solution handles the storage of temporary events at different levels. The first level is the trace buffer, whose size can be modified to conform with the user's

[4]Although we use a specific binary format, we provide a tool that converts trace files in this format into a text format for further use.

needs. For instance, it can be big enough to store all events, thus avoid the necessity of flushing its contents into its corresponding trace file on disk during the execution. The second level is the temporary trace file, which is stored in a temporary file storage facility (tmpfs) available on many Unix-like operating systems[5]. Everything stored in tmpfs is temporary, *i.e.*, in RAM, so there is no I/O operation on disk when the buffer is flushed into the temporary trace file.

Our default approach is to store temporary trace files in /tmp until the end of the execution. After that, they are copied on disk (third level) to avoid I/O operations during the execution of the applications. However, the user can specify a maximum number of flush operations on the trace temporary file. In this case, when the maximum number of flush operations is achieved, the current thread forks a subprocess that will be responsible for copying the temporary trace file to the hard disk. This can be used to limit the amount of RAM memory used by the tracer.

3.5 Case studies: STAMP applications

In this section, we apply our tracing mechanism to gather useful information from TM applications. The goal is to use simple visualization techniques to analyze and understand the behavior of TM applications from traced data. Thus, it can help us to understand the performances obtained in Section 3.2.

In this thesis we are interested in time-based behavior information graphs based on committed and aborted transactions. Since we register the timestamp of each event, we can know when each event occurred during the execution of the TM application. This is helpful to find points of high contention during the execution or to see if the contention is spread out over the whole execution.

To analyze the behavior of TM applications in terms of committed and aborted transactions, we need to register the events $TxEnd$ and $TxAbort$. We believe that this reduced set of events allows us to gather relevant information about TM applications without increasing the degree of intrusiveness.

Since in this thesis we analyze the overall behavior of TM applications, we do not need to obtain a 100% accurate global order of events. For that reason, we

[5]Many Unix distributions enable and use tmpfs by default for the /tmp directory.

used the TSC to manage timestamps due to its negligible overhead. However, to reduce the chances of TSC drift during the experiments, we disabled the dynamic frequency scaling (CPU throttling) of the multicore machine. All experiments were executed on the SMP-24 (a detailed information about this multicore machine is given in Section 5.1).

In the following sections, we discuss in more details the traced information obtained from three STAMP applications: intruder, genome and labyrinth. We selected these applications because they present different levels of contention, different performances and distinct behaviors depending on the STM system used. Our analyses are based on traced information using two state-of-the-art STM systems, *i.e.*, TinySTM and SwissTM. These STM systems presented the best performances among the four STM systems evaluated in Section 3.2.

We implemented the function wrappers for TinySTM as we described in Section 3.4.1. Function wrappers for SwissTM were implemented analogously.

3.5.1 Intruder

In our first case study, we want to understand the causes of the poor the performance of intruder with high thread counts. In addition to that, we want to know why SwissTM performed fairly better than TinySTM.

Figure 3.9 compares the commit rate of transactions in intruder with 2, 4, 8 and 16 threads on TinySTM and SwissTM. The commit rate of an interval $[a; b]$ is given by the Equation 3.1, where $occur(e)_{a;b}$ represents the number of occurrences of the event e in the interval $[a; b]$.

$$Commit\ rate_{a;b} = \frac{occur(TxEnd)_{a;b}}{occur(TxEnd)_{a;b} + occur(TxAbort)_{a;b}} \tag{3.1}$$

Intruder executes a very large number of transactions. If we consider both committed and aborted transactions, intruder executes approximately 255 millions of transactions with 16 threads, which weighs 4GB of traced data in our binary format. In order to reduce the number of points in the graph, we set the intervals to 10

millions of CPU cycles. Points represent commit rates for each interval from the first to the last executed transaction.

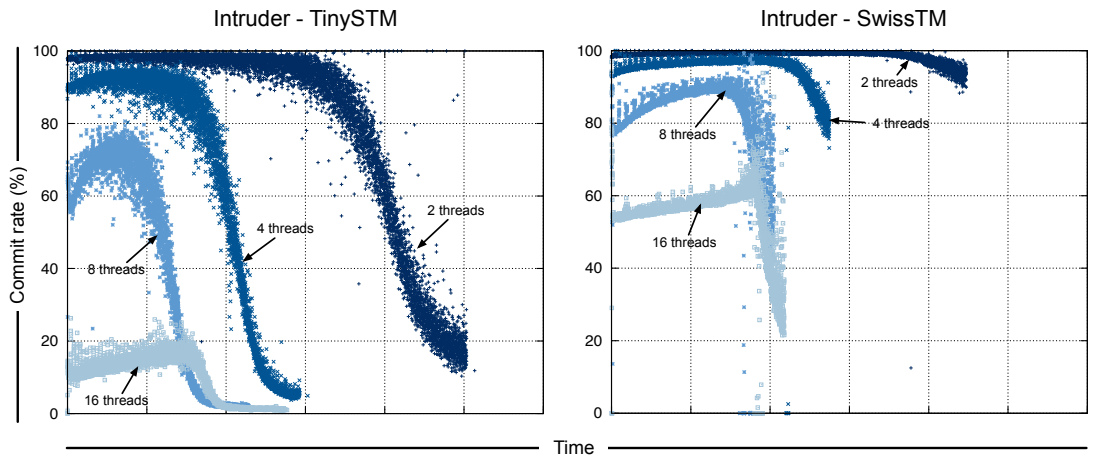


Figure 3.9: Instantaneous commit rates of intruder with TinySTM and SwissTM.

The first conclusion that can be extracted from Figure 3.9 is that the commit rates considerably change during the execution and they are also strongly related to the number of threads in both STM systems. For instance, the commit rate was fairly close to 100% during half of its the execution time with 2 threads on both STM systems. With high thread counts, the commit rate drops to 20% as the execution comes close to end. This indicates a high contention near the end of its execution, which is caused by the fact that there are many threads operating on much less nodes inside the self-balancing tree during the reassembly phase on intruder. Because of that, the probability of having transactions accessing the same nodes is very high.

The second point we observed is that SwissTM presented better commit rates regardless of the number of threads when compared to the corresponding results with TinySTM. With low thread counts, most of time the commit rates observed with SwissTM were higher than 80% even near the end of the execution. Another important observation is that with higher thread counts, SwissTM still have most of the commit rates higher than 60%. An exception occurs with 16 threads, in which the commit rate drops to 20%. However, this is still much better than TinySTM, whose commit rates with 16 threads were never higher than 20%. Those observations explain the fact that intruder presented better performance with SwissTM.

We believe that the reason why SwissTM outperformed TinySTM on intruder is due to the fact that TinySTM uses encounter-time locking as its default scheme. This scheme immediately aborts transactions that try to read a memory location locked by another (writer) transaction. Thus, although read/write conflicts can often be handled without aborts, with this scheme they are detected very early and resolved by aborting readers. In addition to that, TinySTM uses suicide as its default contention management strategy. Suicide performs well on low-contention workloads, which is not the case of intruder. In high-contention workloads, those aborted transactions are restarted right after the conflict is detected and are doomed to abort several times before committing successfully. Contrary to TinySTM, SwissTM detects read/write conflicts lazily, which can allow more parallelism between transactions.

3.5.2 Genome

In our second case study, we want to confirm if the good scalability of genome comes from the fact that it has a constantly high commit rate during the whole execution. In addition to that, we would like to gain some insight on why SwissTM performed better than TinySTM.

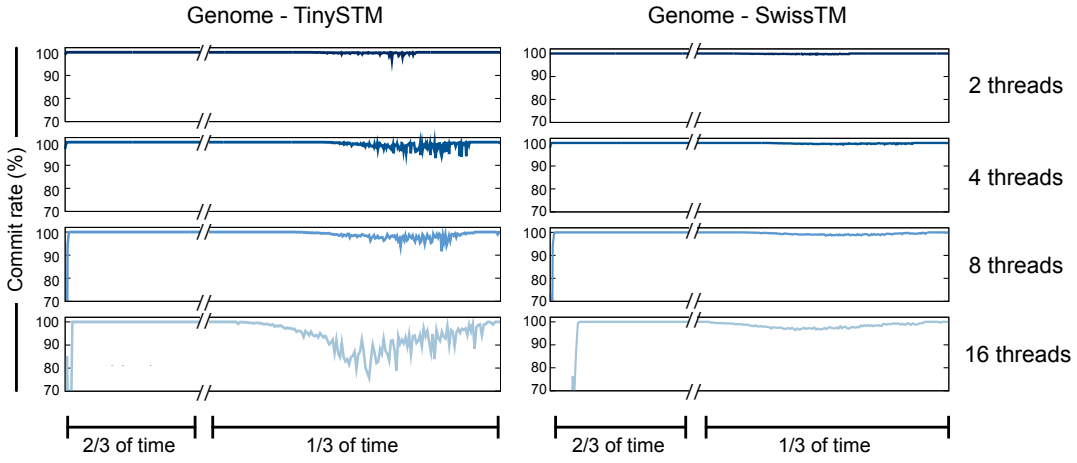


Figure 3.10: Instantaneous commit rates of genome with TinySTM and SwissTM.

Figure 3.9 compares the commit rates of transactions in genome with 2, 4, 8 and 16 threads on TinySTM and SwissTM. Since genome executes much less transactions than intruder (approximately 2 millions of transactions, which weights 140MB of

traced data in our binary format), we use shorter intervals to compute the commit rates (1 million of CPU cycles in this case). We connected the commit rate points with lines to ease the visualization (the commit rates did not vary considerably during the whole execution).

With both STM systems, the commit rate was fairly 100% during the first 2/3 of its execution time, which means that genome has constantly very few aborts. However, we noticed that the commit rate changes at the last third of its execution time. If we compare the commit rates with SwissTM and TinySTM in this period, we can see that it is almost constant in SwissTM with a slightly degradation. In TinySTM, on the other hand, the commit rate considerably varies in the same period. This variation becomes higher as we increase the number of threads. Since genome has a very short execution time (about a second with 16 threads), this variation reduces the performance obtained with TinySTM.

In genome, more than half of all executed transactions are read-only transactions (genome performs about 2.5 millions of commits where 1.5 millions are derived from read-only transactions). We believe that the lower variation in the commit rate observed in SwissTM is due to the fact that SwissTM has a special two-phase contention manager. This contention manager incurs no overhead on read-only transactions and short read-write transactions while favoring the progress of transactions that have performed a significant number of updates. In addition to that, transactions aborted due to write/write conflicts wait for a backoff period proportional to the number of successive aborts, hence reducing the contention in the last third of the execution time in genome. Contrary to SwissTM, TinySTM uses the suicide strategy, which restarts aborted transactions immediately. This increases considerably the variation in the commit rate on the higher-contention period.

3.5.3 Labyrinth

In our third case study, we intend to gain some insights about the performance of labyrinth on TinySTM and SwissTM. As we observed in Figure 3.3, the performances of labyrinth with TinySTM and SwissTM were quite equal for low thread counts. However, labyrinth performed better with TinySTM with 16 threads.

In Figure 3.11, each curve represents the cumulative number of events ($TxEnd$ or

$TxAbort$) during the execution of labyrinth with 16 threads with both STM systems. This metric is calculated as follows. We have two separate counters, one of each event. We then traverse the merged trace file, searching for $TxEnd$ and $TxAbort$ events. Each time we find one of these events, we increment its corresponding counter and we plot this new updated value in the y-axis, using its timestamp in the x-axis. We perform this process to compute the cumulative number of $TxEnd$ and $TxAbort$ events for TinySTM and SwissTM.

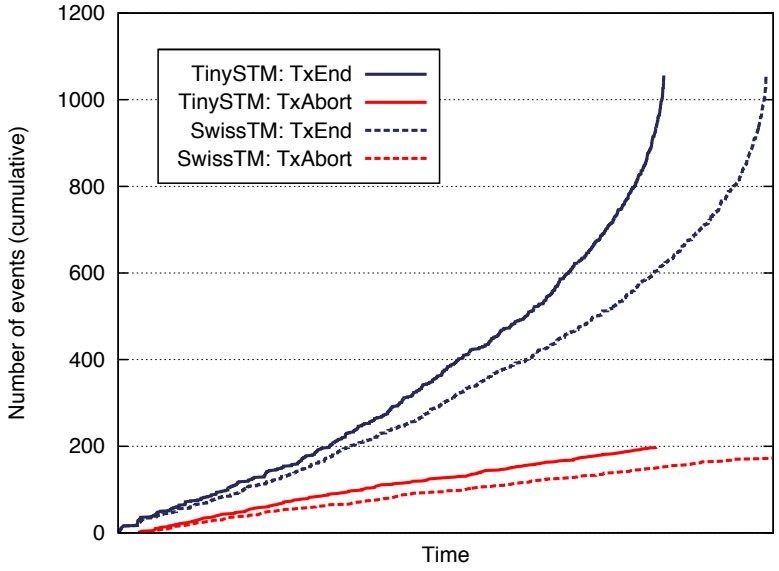


Figure 3.11: Cumulative number of $TxEnd/TxAbort$ in labyrinth with 16 threads.

This graph allows us to identify the growth rate of the events $TxEnd$ and $TxAbort$ during the execution time of the TM application. As it can be observed, the growth rate of $TxEnd$ is exponential and it is very similar on both STM systems. Differently, the growth rate of $TxAbort$ is linear but also very similar on both STM systems. This indicates that labyrinth takes advantage of the TM optimistic approach and justifies its good performance and scalability showed in Figure 3.3.

Although the growth rates of those events are very similar in both STM systems, TinySTM presented better performance than SwissTM. The reason for that stems from the fact that the cumulative number of $TxEnd$ grows faster in TinySTM, indicating a higher throughput in TinySTM (number of committed transactions per unit of time) than SwissTM. Although the $TxAbort$ grows a little faster in TinySTM, it does not

impact too much on the overall performance. Moreover, there are very few read-only transactions in labyrinth: most of the aborts are derived from write/write conflicts. Since the contention is very low in labyrinth, it seems that the backoff period applied by SwissTM on write/write conflicts causes a performance slowdown.

3.6 Concluding remarks

In this chapter, we intended to show that the performance of TM applications can be impacted by several factors. We showed that the STM system plays an important role on the performance of the applications. Since conflicts are transparently solved by the underlying STM system, it may be not trivial to understand the performance obtained from TM applications with a specific STM system.

Some STM systems maintain simple statistics about the number of commits and aborts after the execution of the TM application. However, this only shows to users a global view of the application behavior. In some cases, it is necessary to obtain more detailed information to understand and improve the performance of TM applications or STM systems.

We believe that users can gain some insight on these issues by recording a chronological history of events, representing the application behavior. We thus proposed a tracing mechanism capable of collecting the most interesting events derived from TM applications: $TxBegin$, $TxEnd$, $TxAbort$, $TxRead$, $TxWrite$. Such mechanism can be used with different TM applications and STM systems without any changes in their original source codes. The traced data can then be visualized afterwards using different techniques. In this thesis we were interested in time-based behavior information graphs based on committed and aborted transactions. This is helpful to find points of high contention ("hot spots") during the execution or to see if the contention is spread out over the whole execution. In the former case, developers can try to focus on those hot spots and thus try to improve performance.

We also demonstrated the usefulness of our tracing mechanism with three TM applications from STAMP. Our study was based on the comparison of the traced data from those applications on two state-of-the-art STM systems. We showed that the traced information could help us to better understand the performances obtained.

The information collected during the execution of TM applications was only processed and visualized in a post-mortem manner so far. Differently from that, we can also collect runtime information about the application behavior and use such information to perform some action at runtime. However, runtime analysis usually requires a small amount of storage and frequently relies on sampling techniques to reduce the overhead. This is discussed in the next chapter, in which we propose to perform runtime analysis to increase the performance of TM applications.

CHAPTER 4

Improving the Performance of TM Applications on Multicores

ALTHOUGH we have noticed several works that aim at improving the performance of TM systems, none has given special attention to the performance improvements that can be obtained by matching the characteristics of the TM application and STM system to those of the underlying platform. In this chapter, we tackle this subject through the exploitation of the memory hierarchy of modern multicore platforms. We first demonstrate that the performance of TM applications and STM system can be impacted by the way threads are mapped on cpu cores (Section 4.1) and then we show that it can be complex to determine the best mapping. To tackle this problem, we propose in Section 4.2 a machine learning-based approach to automatically infer suitable thread mapping strategies for TM applications. Then, we show how this approach can be used statically (Section 4.3) and dynamically (Section 4.4). Finally, in Section 4.5, we conclude this chapter.

4.1 Impact of thread mapping on TM applications

In parallel applications, threads must cooperate in order to accomplish a required computation. However, as discussed in Section 2.1, the latency or memory contention observed from threads may be different depending on which cores they are executing

and how the memory hierarchy and interconnection are used. One technique to deal with that is called thread or processes mapping, which aims at mapping threads or processes to specific cores to improve the use of resources such as interconnections, main memory and cache memories.

Defining an efficient thread mapping strategy for parallel applications is challenging. It may be simpler on regular parallel applications such as the ones implemented with OpenMP [Cas+09]. On those applications, the access pattern may be regular and determined statically before the application execution. This facilitates the burden of choosing an efficient thread mapping strategy. Indeed, finding patterns of regular applications to map threads on cores has been studied in several contexts. However, this has not been studied in the context of TM applications, which exhibit irregular access patterns.

On TM applications, determining a suitable thread mapping strategy for a specific TM application/STM system/platform configuration is even more challenging. This stems from the fact that each STM system implements different mechanisms to detect and solve conflicts between transactions. Those mechanisms modify the behavior of the TM application, so the best thread mapping strategy is also affected.

To support this fact, we performed several experiments with all TM applications available from STAMP. Again, we used the largest input sizes for all applications and low contention parameters for kmeans and vacation as described in the STAMP's original paper [Min+08]. We ran all applications using 8 threads with four different STM systems (TinySTM, SwissTM, TL2 and RSTM) and four thread mapping strategies (compact, scatter, round-robin and linux). We illustrate these strategies in Figure 4.1. Basically, these strategies can be characterized as follows:

- **scatter**: threads are distributed across different processors. This avoids cache sharing between cores in order to reduce the contention on the same cache. This strategy is beneficial to applications whose threads usually access disjoint data.

- **compact**: threads are physically placed on sibling cores. This reduces memory access latency, since threads share all levels of the cache hierarchy. This strategy is beneficial to applications whose threads usually access the same amount of data.

- **round-robin**: it is an intermediate solution in which threads share higher levels of cache (*e.g.*, L3) but not the lower ones (*e.g.*, L2). This strategy can only be applied when the platform has more than one L3 shared caches.
- **linux**: it is the default Linux scheduling strategy. It is a dynamic priority-based strategy that allows threads to migrate to idle cores to balance the run queues.

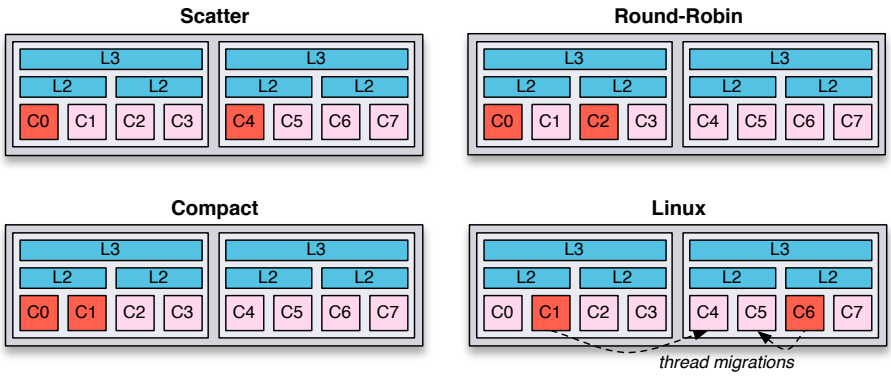


Figure 4.1: Thread mapping strategies.

Table 4.1 shows the results we obtained after executing all applications with different STM systems and thread mapping strategies. We carried out these experiments on the SMP-24 platform (Section 5.1 presents a detailed specification of this multicore platform). We present the *best* and *worst* thread mapping strategies for each pair of application and STM system along with their execution times (in seconds). Results represent mean execution times of at least 30 executions with a maximum standard variation of 2%.

As shown in Table 4.1, the difference between the best and worst thread mappings execution times can be high depending on the application and STM system.[1] For instance, consider the application **vacation** executed with **RSTM** and **SwissTM**. When vacation was executed with RSTM, the best strategy was compact, which was approximately 1.4 times faster than linux (the worst strategy). Surprisingly, compact was the worst strategy for the same application running with SwissTM, which was approximately 1.2 times slower than scatter (the best strategy). This

[1]We did not include the results of bayes with TL2, since this configuration led to incorrect executions of this application.

stems from the fact that each STM system implements different mechanisms to detect and solve conflicts between transactions. Those mechanisms modify the behavior of the application, so the best thread mapping strategy is also affected.

Not surprisingly, we also observed variations in terms of best and worst thread mapping strategies within the same STM system while executing different applications. As an example, we can cite the applications **kmeans** and **labyrinth** on TinySTM. The best strategy for kmeans was compact (approximately 1.2 times faster than scatter) whereas the best one for labyrinth was scatter (approximately 1.3 times faster than round-robin).

Application	TinySTM				SwissTM			
	Best mapping		Worst mapping		Best mapping		Worst mapping	
	name	*time (s)*	*name*	*time (s)*	*name*	*time (s)*	*name*	*time (s)*
bayes	compact	6.8	scatter	8.9	scatter	6.1	compact	9.6
genome	compact	4.0	scatter	10.2	scatter	1.5	compact	1.9
intruder	compact	44.9	linux	65.7	compact	15.7	scatter	16.9
kmeans	compact	6.7	scatter	7.8	round-robin	5.7	compact	6.6
labyrinth	scatter	15.6	round-robin	20.3	scatter	14.5	round-robin	15.7
ssca2	scatter	7.3	compact	9.6	scatter	7.5	compact	10.0
vacation	round-robin	7.7	linux	10.7	scatter	8.0	compact	9.8
yada	compact	13.1	scatter	17.1	compact	11.2	scatter	14.1

Application	TL2				RSTM			
	Best mapping		Worst mapping		Best mapping		Worst mapping	
	name	*time (s)*	*name*	*time (s)*	*name*	*time (s)*	*name*	*time (s)*
bayes	–	–	–	–	compact	7.8	round-robin	8.8
genome	scatter	2.2	compact	2.6	compact	2.5	linux	2.8
intruder	compact	34.5	scatter	39.6	compact	25.5	scatter	34.9
kmeans	round-robin	7.1	scatter	8.4	compact	11.8	scatter	17.2
labyrinth	round-robin	22.9	scatter	25.7	scatter	17.6	compact	19.6
ssca2	compact	11.8	scatter	15.3	compact	20.7	scatter	35.6
vacation	scatter	9.5	compact	10.4	compact	9.7	linux	13.3
yada	compact	16.3	scatter	20.0	compact	36.9	scatter	45.8

Table 4.1: Impact of thread mapping strategies on TM applications.

We also executed all those experiments on another platform that has a different memory hierarchy. We concluded that the platform also plays an important role. This means that the best thread mapping strategy for same TM application/STM system configuration may not be the same on all platforms. Thus, it is a hard problem to determine a suitable thread mapping strategy for a TM application considering both STM system and platform characteristics. In the next section, we present our solution to tackle this problem. We propose a machine learning-based approach to

predict thread mapping strategies based on the STM system, application and platform characteristics.

4.2 A machine learning-based approach for thread mapping

Machine Learning (ML) is a scientific discipline that is concerned with the design and development of algorithms that allow computers to evolve behaviors based on empirical data [Cza11; Mit97]. We believe that ML can be used in our context to deal with the complex relations between TM applications, STM systems and platforms and serve as an heuristic for predicting efficient thread mapping strategies while taking into consideration the characteristics of the TM applications, STM systems and platforms.

4.2.1 Overview of the ML-based approach

Our proposal stems from the fact that ML can be useful to model the behavior of complex interactions between applications, systems and platforms. Then, it can provide a portable solution to predict the behavior of new combinations of TM applications and STM systems, also called **instances**, based on *a priori* profiled runs.

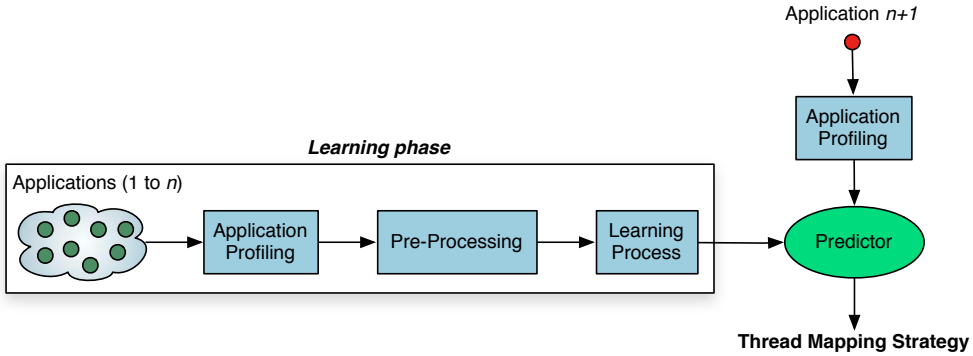


Figure 4.2: Overview of our ML-based approach.

The core of our ML-based approach is called the *learning phase*. The learning phase is subdivided in the following three major steps: application profiling, data

pre-processing and learning process. The overview of our ML-based approach is illustrated in Figure 4.2.

In the application profiling step, each instance of TM application/STM system has to be run and profiled. This profiling step involves gathering data of several characteristics, also known as **features**, available in the STM system and platform. Then, in the pre-processing step, a database of profiled instances is built up and pre-processed in order to remove duplicated information, to normalize or to convert values. In our approach, groups of instances that share the same feature values are merged to a single input instance sample where the target variable is the most suitable thread mapping strategy based on a ranking. Finally, in the learning process step, we use a ML algorithm that outputs a predictor. Once the learning phase is finished, new instances can be profiled and then the collected data is used as an input to the predictor, which will be capable of inferring an efficient thread mapping strategy (which is our **target variable**) for the new unknown instance.

In the following sections, we discuss each one of the three steps of our approach.

4.2.2 Application profiling

In the first step of the learning phase, we have to select which features are relevant to be profiled. Table 4.2 presents the selected features classified into three categories as well as the target variable.

Category	Feature	Values	Description
A	Tx Time Ratio	(0.0; 1.0)	Tx time / Execution time
	Tx Abort Ratio	(0.0; 1.0)	Tx aborts / Tx commits + Tx aborts
B	TM Conf. Detection	eager, lazy	STM conflict detection policies
	TM Conf. Resolution	suicide, backoff	STM conflict resolution policies
C	LLC Miss Ratio	(0.0; 1.0)	Cache misses / Total accesses
T	Mapping Strategy	compact, scatter, round-robin, linux	Thread mapping strategies

Table 4.2: TM application, STM system features and the target variable.

Features from **Category A** summarize the interaction between the application and the STM system. These selected features are commonly available across different STM implementations. Both *transaction time ratio* and *abort ratio* can be extracted and computed from execution time and STM statistics.

STM systems can implement more than one conflict detection and resolution mechanisms to handle concurrent transactional executions. Because of that, we also consider those mechanisms as input features. More precisely, we consider two possibilities for the conflict detection (eager and lazy) and resolution (suicide and backoff). The reason for that is that those mechanisms are frequently used and they are usually available in most of STM systems. **Category B** resumes these features for both conflict detection and resolution.

In **Category C**, we observe the *Last Level Cache (LLC) miss ratio* to represent the interaction between the application and the platform. In order to measure the LLC miss ratio, it is necessary to access hardware counters. Tools such as the Performance Application Programming Interface (PAPI) [Ter+10] provide an interface to access such information.

Last, Table 4.2 also presents the thread mapping strategies for our target variable (**T**) which our ML-based approach aims to predict. The selected strategies are the ones explained in Section 4.1.

4.2.3 Data pre-processing

The accuracy of a ML prediction model relies upon the quality of the input data samples. Thus, raw profiled data has to be pre-processed before feeding a ML algorithm. Our first step is to normalize or convert features according to the selected ML algorithm. As we explain in the next section, our selected learning algorithm works exclusively with categorical or discrete features. Since conflict detection and resolution mechanisms are already discrete, they do not need to be converted. For the other features, we apply a simple discretization technique to convert each ratio value within a range $(x; y)$ into one of the three following discrete values: i) *low* $(0.0; 0.33)$; ii) *medium* $(0.33; 0.66)$; and iii) *high* $(0.66; 1.0)$. In order to reduce ambiguity among input data samples, it is still possible to increase the number of features and ranges. Although there exists more complex discretization techniques,

we show in Section 5.2 that the selected features and ranges are enough to obtain accurate predictions.

After data conversion, the second step is to determine the target variable value. In our approach, we aim to predict the most efficient thread mapping strategy for an application/STM/platform configuration. For this purpose, for each group of input instances that shares the same feature values (*i.e.*, same TM application/STM system configuration) but with different thread mapping strategies, we select the most suitable strategy based on the application execution time. Then, each group of instances are merged and becomes a new single input instance in which the target variable is the mapping strategy that presented the lowest execution time.

Frequently, the execution time of different mapping strategies for the same application and STM configuration can be statistically equivalent. In that case, the decision of which strategy is the most suitable becomes non-trivial. From the ML algorithm perspective, ambiguous input instances should be avoided. For example, if strategy A is 0.1% faster than strategy B in one case and the opposite in another similar case, we should decide to elect just one strategy as the "fastest one" in both cases. Otherwise, the ML algorithm is not able to identify a common pattern on instances with similar behavior. Although the instances share similar feature values, they use different thread mapping strategies.

In order to address this ambiguity in our data samples, we propose an algorithm to rank the strategies before resolving these ties. This algorithm introduces an error percentage threshold ε to distinguish mapping strategies that achieve very similar performance.

Firstly, for each group of instances that shares the same feature values except the target variable value, a mapping strategy can only be claimed as the best one if its execution time is at least ε less than all the other ones. Secondly, for each thread mapping strategy, our algorithm counts how many times it was classified as the best one. This provides a ranking to distinguish all the other instances within a group that presented statistically similar performance. Thus, thread mapping strategies with higher counts have higher priority. Finally, we resolve all these ties by assigning to the target variable within a group the most suitable thread mapping strategy based on that ranking. This avoids having very similar input instances present different outcomes. Once the input data samples are pre-processed, our approach moves onto

the learning phase.

4.2.4 Learning process

There exist several machine learning algorithms, including Neural Networks [Zha00], Support Vector Machines (SVMs) [CS00] and Decision Tree Learning [Qui86]. Typically, a machine learning algorithm takes as input a data set S of input instances containing values for each corresponding feature in F, including the target variable. It applies an iterative process on S to adjust its internal modeling parameters until it fits to the input data. This process is also known as the *training* phase. Then, it outputs a predictor (*e.g.*, a set of functions, rules, etc.) that infers the target variable for a new unknown input instance.

Decision tree learning is one of the most widely used and practical methods for inductive inference. It is a method for approximating discrete values, in which the learned function is represented by a decision tree [PK01]. A decision tree classifies instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance (target variable).

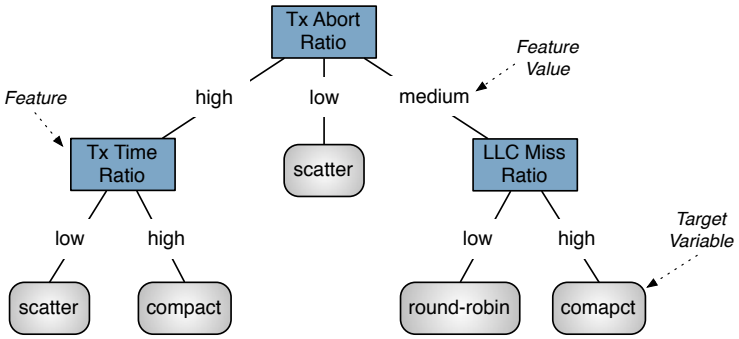


Figure 4.3: Hypothetical example of a decision tree.

Figure 4.3 shows a hypothetical example of a decision tree that considers some of the features presented in Section 4.2.2. Each internal node of the tree represents an input feature. In this example, the input features considered are *transaction abort ratio*, *transaction time ratio* and *last-level cache miss ratio*. Edges connecting one internal node to its successors are labeled with one of its feature values (in this example, *low, medium* and *high*). Successor nodes can be either a feature or a value

of the target variable (thread mapping strategy). If the node contains the latter, it means that it is a leaf node.

We decided to use decision tree learning as the learning algorithm to construct the thread mapping predictor. The reason for that is fourfold: (i) STM features such as conflict detection and resolution policies are inherently discrete; (ii) the profiled data are usually continuous numerical values but they are more representative if grouped into discrete values (in fact they can be easily converted to discrete values using simple discretization techniques as we showed in the previous section); (iii) decision trees are simple to understand and interpret; and (iv) the predicted model can be easily integrated to any application/system with little overhead. For those reasons, the decision tree learning becomes a suitable learning algorithm to address our problem.

A well-known decision tree learning algorithm is the Iterative Dichotomiser 3 (ID3) [Qui86]. The ID3 learning algorithm outputs a decision tree as a predictor based on categorical discrete input instances. Given a set S of input instances and a set of F features, the ID3 learning process starts choosing the "best" feature to be in the root node. By best, it means the feature that splits S in more homogenous subsets, that is, subsets where the majority of input instances contains the same target variable value. Homogeneity is measured by $Entropy$ defined in Equation 4.1 where p_i is the probability that a given instance is in a subset with the same target variable value.

$$Entropy(S) = \sum p_i \log_2(p_i) \tag{4.1}$$

$Gain$ can be derived from $Entropy$ by computing the subtraction between the entropy of S and the sum of the entropies of its subsets S_i as shown in Equation 4.2. The feature with highest $Gain$ is chosen to be the root node.

$$Gain(S,F) = Entropy(S) - \sum_{i \in F} \frac{|S_i|}{|S|} \times Entropy(S_i) \tag{4.2}$$

This recursive process is repeated for each successor node until all leaf nodes contain only input instances with the same target variable value. The ID3 main algorithm is presented below.

```
input  : Input instances S, Features F
output : Decision tree T
begin
    F_max ← max(Gain(S, F_i));
    Split S into subsets S_j by each value V_j ∈ F_max;
    Create a subtree U where the root node is F_max;
    foreach V_j ∈ F_max do
        Add an edge labeled V_j to U;
        Connect a successor node N_j containing S_j;
        Add to T the subtree U;
        if Entropy(N_j) = 0 then
            Set label of N_j to target variable value;
            Remove S_j instances from S;
        else ID3(S_j, F);
    end
end
```

Function ID3(S, F)

In order to verify the accuracy of the resulting decision tree, it is possible to use the standard *leave-one-out cross-validation* method. To do so, for each input instance, the learning algorithm is trained with the remaining instances and tested with the unused instance. Thus, the result indicates how good is the the decision tree.

4.2.5 Prediction

Once the ID3 algorithm outputs a predictor and its accuracy is verified, it can be used to determine a suitable thread mapping strategy to be applied to new unobserved instances. It is important to notice that the learning phase is no longer required for any new input instance. Instead, profiling is needed in order to gather application and STM system information of the new input instance. The profiled data is converted to conform with the decision tree input format. The decision tree is then fed with the new input instance information and traversed, outputting a thread mapping strategy. The predicted thread mapping strategy can be then applied.

4.3 Static thread mapping

In this section, we show how our machine learning-based approach can be applied to perform static thread mapping on TM applications. Firstly, in Section 4.3.1, we discuss the input instances we used to feed the learning process. Then, in Section 4.3.2, we show the decision trees obtained after applying the ID3 learning algorithm on those input instances. Finally, in Section 4.3.3, we describe how these decision trees can be used to perform static thread mapping on TM applications.

4.3.1 Gathering input data to feed the learning process

Once we defined our ML-based approach for thread mapping on TM applications, it has now to be fed with input instances to construct the predictor for practical usage. This is an important step since the predictor will learn from data collected from the input instances. If the input instances do not represent good samples, the predictor may make many incorrect predictions.

Because of that, we decided to use all applications from the STAMP benchmark suite as our input instances. As previously discussed in Section 2.3.2, the applications from STAMP consist of a variety of algorithms and different application domains. Moreover, they are composed of a wide range of transactional workloads that differ from each other in the size of transactions, the amount of contention and the time spent inside transactions. We believe that those realistic applications can be useful to construct a predictor capable of capturing the behavior of TM applications.

We proceeded as follows to perform the learning process. Firstly, we profiled all STM applications from STAMP to gather the information (features) needed by the ML such as transactional time ratio, transactional abort ratio and LLC miss ratio of each TM application as explained in Section 4.2.2. Then, we measured the execution time of each application while varying the thread mapping strategy (linux, scatter, compact and round-robin) as well as the conflict detection (eager or lazy) and resolution mechanisms (backoff or suicide). We used TinySTM as our target STM system, since it can be configured with different conflict detection and resolution mechanisms, we can have configurations that are similar to other STM systems and even other different configurations.

Figure 4.4 shows the speedups of all possible combinations.[1] We used two different platforms during our experiments. The SMP-16 has a single level of shared caches whereas SMP-24 has two levels of shared caches. The speedups refer to the parallel version of the applications with 8 threads in comparison to the corresponding sequential version. The results represent the mean speedups obtained over a minimum of 30 executions. For a more precise description of the platforms and the speedup metric, please refer to Section 5.1.

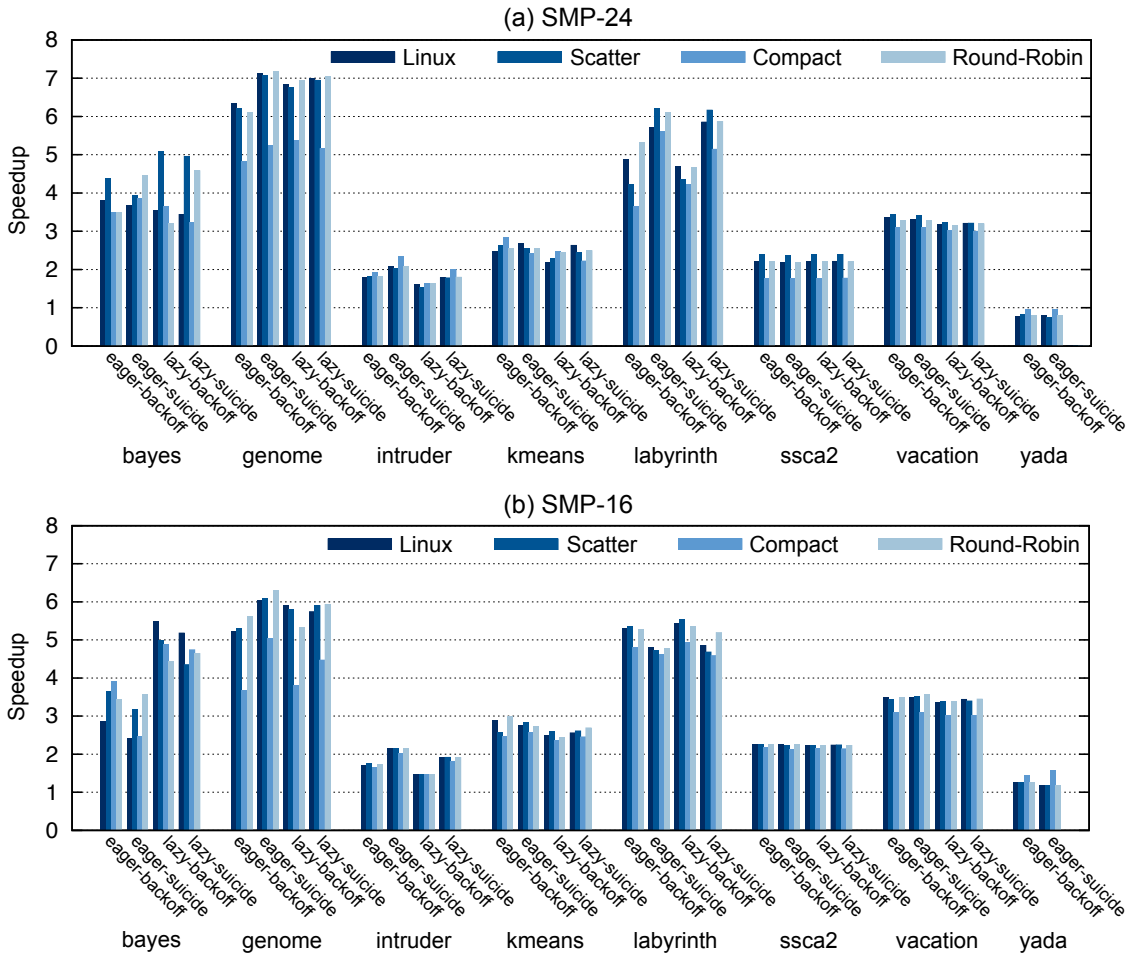


Figure 4.4: Impact of thread mapping strategies on the performance of TM applications with different STM configurations.

As it can be observed, the SMP-24 is more sensitive to thread mapping than the

[1]We omitted the results of yada with lazy conflict detection, since this configuration led to incorrect executions of this application.

SMP-16. Most of the applications presented a more significant variance in terms of speedup when different thread mapping strategies were applied. Specially, a significant impact on the performance can be noticed on **labyrinth** and **ssca2** on the SMP-24 whereas in SMP-16 such impact is much less perceivable. This is explained by the fact that SMP-24 has a more complex memory hierarchy (two levels of shared caches). This leads to more complex performance tradeoffs in terms of memory contention and latency if threads are not mapped appropriately.

We also noticed that some applications are insensitive to the STM system parameters on both platforms. Modifying the conflict detection and resolution mechanisms does not change the performance perceived by these applications. This can be explained by the fact that applications such as **ssca2** do not spend much time within transactions.

In contrast, other applications such as **bayes** and **genome** proved to be very sensitive to both thread mapping strategies and STM system parameters. On these applications, compact usually delivered less performance gains than other thread mapping strategies. This occurs for two reasons. From one perspective, bayes presents a high abort ratio. This means that transactions usually abort several times before committing. Each time a transaction is aborted, all the accesses to the shared data inside the transaction are re-executed. This fact increases the probability of having many transactions being executed at the same time, increasing the contention on cache memories. Since compact forces threads to share cache, such contention is even higher.

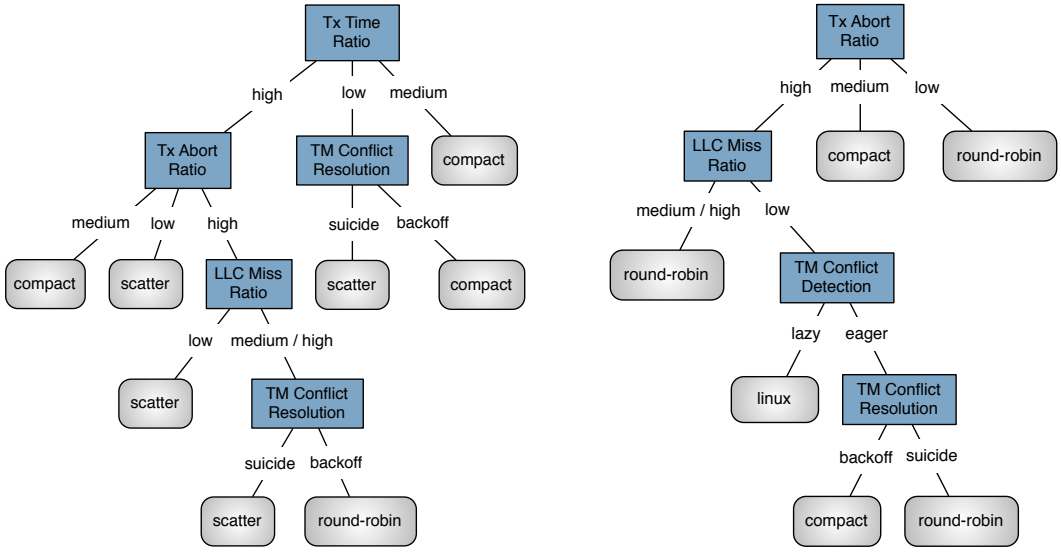
From other perspective, genome is characterized by having a high transaction time ratio and low contention. This means that this application executes transactions most of the time and transactions usually access disjoint data. In this case compact is not beneficial because the same cache will be used to hold disjoint data of several threads. This increases the contention on the cache and reduces the amount of cache available for each thread.

4.3.2 Generating the decision trees

As observed in the previous section, the target platform can influence the impact of the thread mapping strategy perceived by an application/STM configuration.

Because of this, we trained our ML-based approach to predict a suitable thread mapping strategy considering each platform separately.

The profiled information obtained for all possible feature combinations was pre-processed as discussed in Section 4.2.3. The pre-processed data was then used to feed the ID3 learning algorithm. Instead of implementing the ID3 algorithm, we used a machine learning tool called Weka [Bou+10]. It implements several machine learning algorithms including ID3. Weka also provides a cross-validation method to verify the accuracy of the generated decision tree.



(a) SMP-24 (b) SMP-16

Figure 4.5: Decision trees generated by the ID3 learning algorithm on both platforms.

Figure 4.5 shows two decision trees generated by the ID3 learning algorithm on the SMP-24 (on the left side) and SMP-16 (on the right side). It is important to mention that, due to the ranking algorithm in the pre-processing step, some thread mapping strategies can be dropped from the decision tree even when they are actually the best one for a given instance. This occurs when the thread mapping strategy does not present significantly more performance gains than other strategies. That was the case of the linux strategy in the decision tree for SMP-24.

The influence of the target platform is confirmed by the decision trees, which are very different from each other. On the SMP-24, compact and scatter are the two

most important thread mapping strategies whereas round-robin and compact are the most important ones on the SMP-16. We can also observe that the decision tree of the SMP-16 is much simpler than the SMP-24. This can be explained by the fact that the SMP-24 has a more complex cache hierarchy.

Even though there are differences between the decision trees, we can derive some overall conclusions. Firstly, there is a possible correlation between the abort ratio and the LLC miss ratio. On both trees, when an application presents a high abort ratio, the LLC miss ratio is taken into account to decide the thread mapping policy to be applied. Secondly, when the abort ratio is low, the predictor tends to select a strategy that places threads far from each other (*e.g.*, scatter and round-robin strategies). A low abort ratio means that transactions rarely access the same shared data at the same time. Thus, the contention generated by several threads accessing the same cache can be alleviated by applying such strategies. Finally, when the conflict detection used is backoff, the decision tree tends to decide for the compact thread mapping strategy. Backoff forces transactions to wait some time before re-executing due to aborts. This reduces the amount of contention on the cache, making it possible to place threads on sibling cores to amortize the access latency.

An exception is the Linux default strategy that appears only in one specific case. It tends to distribute and migrate threads among cores in order to balance the workload. Thus, the resulting distribution of threads is variable and unpredictable. These characteristics thus benefit applications with low cache miss ratio and very dynamic behavior. This is the case of bayes on SMP-16 in which the lazy detection mechanism contributes to reduce the number of false aborts compared to the eager mechanism.

4.3.3 Predicting and applying thread mapping strategies

Once the predictor has been constructed, it can now predict a suitable thread mapping strategy for new unknown TM applications. We can use two different strategies to apply the predicted thread mapping strategy statically:

- **Prediction after a previously profiled execution**: the unknown TM application is executed once and profiled until it finishes to gather the information needed by the ML. Then, the decision tree is then fed with the new input

instance profiled information and traversed, outputting a thread mapping strategy that is applied on subsequent executions of this TM application.

- **Prediction at runtime after a warm-up profiling period**: the unknown TM application starts running with a default thread mapping strategy and during a initial warm-up interval it is profiled. The profiled data is converted to conform with the decision tree input format. The decision tree is then fed with the new input instance information and traversed, outputting a thread mapping strategy. The predicted thread mapping strategy is then applied at runtime.

Both strategies have pros and cons. The former has much more information about the TM application, since it profiles the whole execution of the application, but it is necessary to execute the application once before re-executing it with the predicted thread mapping strategy. On the other hand, the latter has much less information since the profiling is done during an warm-up interval, but the predicted thread mapping strategy is applied at runtime. If the profiled interval does not correspond to the overall behavior of the application, the prediction can be uncorrected. We evaluate the accuracy of our static thread mapping later on (Section 5.2).

4.4 Dynamic thread mapping

In the previous section, we focused on the used of our ML-based approach to statically infer a suitable thread mapping strategy for TM applications. This means that the predicted thread mapping strategy is applied once at the beginning and does not change during the execution of the application. Static thread mapping can benefit from TM applications whose most of the transactions usually have very similar behavior.

However, we have constantly seen efforts for a wider adoption of Transactional Memory. For instance, the latest version of the GNU Compiler Collection (GCC 4.7) supports TM primitives. In terms of hardware support, we can cite the new BlueGene/Q processors and the Intel Transactional Synchronization Extensions (TSX) for the future multicore processor code-named “Haswell”. Thus, it is expected that more complex applications will make use of TM in a near future. These applications will probably have multiple execution phases with a different transactional behavior

in each phase. In those cases, static thread mapping will no longer improve the performance of those applications, emerging the necessity of a dynamic approach able to identify these different phases and switch to a more adequate thread mapping strategy during the execution.

In the following sections, we explain how our ML-based approach can be used to perform dynamic thread mapping. We also present its implementation within a state-of-the-art STM system.

4.4.1 From static to dynamic thread mapping

In order to adapt our ML-based approach to perform dynamic thread mapping, we propose to extend the previous *static prediction at runtime after a warm-up profiling period*. The idea is to perform the profiling and prediction several times during the execution of the TM application, adapting the thread mapping strategy to the current workload behavior whether necessary. Since most of the considered characteristics can vary during the execution of applications composed of several phases, it is important to define how and when the profiling will occur. We propose to use sampling instead of profiling the whole execution of the application. This reduces considerably the profiling overhead. The interactions between the profiler and the application are illustrated in Figure 4.6.

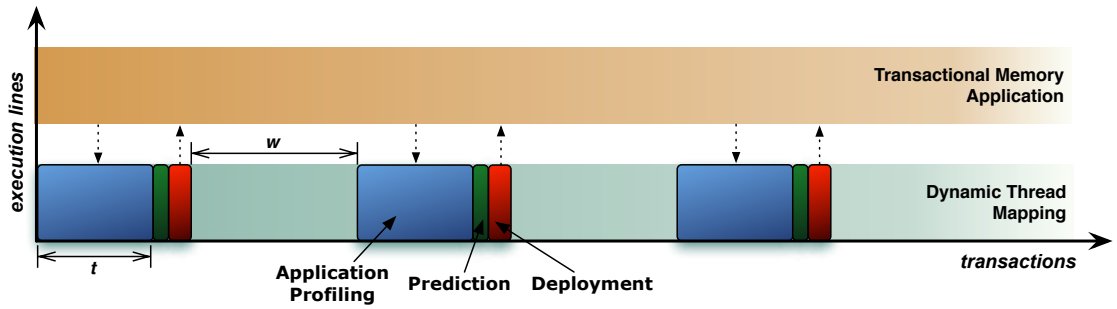


Figure 4.6: Application execution with dynamic thread mapping.

During the TM application initialization, we set the default thread mapping strategy to linux. Then, we profile t committed transactions at runtime to gather the information needed by our ML-based predictor (represented in Figure 4.6 as blue boxes). The profiled information is used by the ML-based predictor to select a thread

mapping strategy that suits the current workload characteristics (represented in Figure 4.6 as green boxes). Then, the predicted thread mapping strategy is deployed (represented in Figure 4.6 as red boxes) and remains unchanged during w committed transactions. This process is repeated until the TM application ends.

The parameters t and w are specified by the number of committed transactions instead of time. This guarantees that our measures occur when transactions are being executed. They can be fixed or can be adapted during the execution. To adapt these parameters during the execution, we may use a hill-climbing strategy. In this strategy, we start with short periods and we double them each time the predicted thread mapping strategy was not changed. This is done until a maximum interval size is reached. When the thread mapping strategy is changed due to a phase transition, we reset them to their initial values and the hill-climbing strategy is restarted.

4.4.2 Implementation on TinySTM

For our solution to be transparent to users, we decided to implement it within an STM system. We chose TinySTM among other STM systems because it is lightweight, efficient and its implementation has a modular structure that can be easily extended with new features. Figure 4.7 shows an overview of TinySTM.

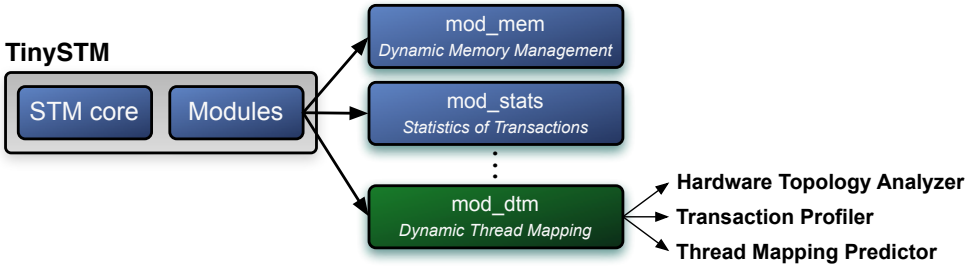


Figure 4.7: Implementation of our dynamic thread mapping in TinySTM.

Basically, TinySTM is composed of an STM core, in which most of the STM code is implemented, and some additional modules. These modules implement basic features such as the dynamic memory management (`mod_mem`) and transaction statistics (`mod_stats`). We added a new module called `mod_dtm` that extends TinySTM to perform dynamic thread mapping transparently.

Our module combines the following tree main components:

- **Hardware topology analyzer** is responsible for gathering useful information from the underlying platform topology (*i.e.*, the hierarchy of caches and how they are shared among the cores). Such information is important to correctly apply the thread mapping strategies. However, such information is usually obtained from system-specific operations, such as reading the pseudo-filesystem on Linux, or calling specific low-level libraries on other operating systems. In order to avoid those system-specific operations, we decided to use the Hardware Locality (hwloc) library [Bro+10]. This library exposes a portable abstracted view of the hardware topology to the runtime systems and works on several operating systems.

- **Thread mapping predictor** relies on the decision tree shown in Figure 4.5 to predict the thread mapping strategy. During the prediction phase, the tree is traversed using the profiled information from the *transaction profiler* and the resulting thread mapping strategy is then deployed.

- **Transaction profiler** performs the application runtime profiling to gather information from hardware counters and transactional basic statistics. Its pseudo-code is depicted in Figure 4.8. The *cache miss ratio* is obtained through PAPI [Ter+10] to access hardware counters. We maintain two counters to calculate the *abort ratio* (named Aborts and Commits). The *transactional time ratio* is an approximation obtained by measuring the time spent inside and outside transactions.

TinySTM allows the inclusion of user-defined extensions. In our case, we instrumented three basic TM operations (Figure 4.8) that are called when transactions start (`start`), when they are rollbacked in case of conflicts (`abort`) and when they finish successfully (`commit`). Thus, every call to these operations is intercepted by our module, which executes the *transaction profiler* during the profilling periods and calls the *thread mapping predictor* to switch the thread mapping strategy when necessary.

In order to reduce the intrusiveness on the overall system (*i.e.*, to not change the behavior of the application) and to avoid extra synchronization mechanisms to guarantee reliable measures among concurrent threads, we assume that the workload

```
// on transaction start
if is profiling period then
    if first tx in this period then
        StartPapi(LLCAccess, LLCMiss);
        ProfileTime ← GetClock();
    end
    TxTime ← GetClock();
end

// on transaction abort
if is profiling period then
    Aborts ← Aborts + 1;
end
```

```
// on transaction commit
if is profiling period then
    TxTime ← GetClock() − TxTime;
    TotalTxTime ← TotalTxTime + TxTime;
    Commits ← Commits + 1;
    if last tx in this period then
        StopPapi(LLCAccess, LLCMiss);
        ProfileTime ← GetClock() − ProfileTime;
        TotalNonTxTime ← ProfileTime − TotalTxTime;
        ThreadMapping ← TMPredictor();
        ResetAllCounters();
    end
end
```

Figure 4.8: Transaction profiler pseudo-codes.

of TM applications is uniformly distributed among the threads. This means that profiling a single thread during a certain period can give us a good approximation of what would be captured if all threads were profiled during the same period. This assumption simplifies the profiling process, since we can use only one thread for profiling the TM application at a time. In our implementation, when a TM application is executing, only one thread among all concurrent threads is chosen to be the *transaction profiler*.

4.5 Concluding remarks

As we previously discussed in this thesis, the performance of TM applications can be improved at different levels, ranging from the TM application to the underlying platform levels. Diverging from several previous works that focused on improving the performance on a single level, we showed in this chapter that the performance of TM applications can be improved if we match their characteristics (along with the characteristics of the STM system) to the underlying multicore platform. More precisely, we were interested in gathering useful information from the TM applications and STM systems and then use such information to better exploit the memory hierarchy of modern multicore platforms.

One technique to deal with that is called thread mapping, which aims at mapping threads to specific cores of the underlying platform to improve the use of resources

such as interconnections and cache memories. However, the impacts of applying thread mapping on TM applications had not been explored before. In addition to that, STM systems make this task even more difficult due to the runtime system. Existing STM systems implement several conflict detection and resolution mechanisms, which leads TM applications to behave differently for each combination of these mechanisms. The result is that it is not trivial to predict a suitable thread mapping strategy for a specific TM application/STM system/platform.

We tackle this problem by using Machine Learning to automatically infer a suitable thread mapping strategy for TM applications. Our proposal stems from the fact that ML can be useful to model the behavior of complex interactions between applications, systems and platforms. It is based on a prior learning phase, in which we profile several TM applications running on different STM systems to know their characteristics. Then, such information is used to feed a ML algorithm that outputs a predictor. Once this learning phase is finished, new instances can be profiled at runtime and the collected data can be used as an input to the predictor, which will be capable of inferring an efficient thread mapping strategy for the new unknown TM application.

The predictor confirmed our intuitions pointed out in Section 1.1.2. Placing threads on sibling cores usually improves the performance of TM applications whose transactions have a moderate to high conflict probability (*i.e.*, they usually access the same amount of shared data). In those cases, our predictor usually decides to apply the compact strategy. In an opposite situation, distributing threads across different processors (thus avoiding cache sharing) usually reduces the contention on the same cache and is beneficial to TM applications whose transactions access a large amount of disjoint data, thus rarely conflicting. In those cases, our predictor usually decides to apply either scatter or round-robin strategies.

Once the predictor is constructed, it can be used to perform static and dynamic thread mapping. In the static approach, we can either predict the thread mapping after a previously profiled execution of the unknown TM application or during the execution after a warm-up profiling period. In both strategies, the predicted thread mapping is applied and remains unchanged during the whole execution. In the dynamic approach, on the other hand, we used profiling techniques to gather useful information during certain periods of the execution of TM applications, switching the

thread mapping strategy to a more adequate one at runtime when necessary. Both approaches were implemented in TinySTM, a state-of-the-art STM system, which makes static/dynamic thread mapping transparent to the user.

CHAPTER 5

Experimental Evaluation

THIS chapter presents the experimental evaluation of the static and dynamic thread mapping solutions proposed in this thesis. We start by describing our experimental setup (Section 5.1). Then, we analyze the performance improvements obtained with both static (Section 5.2) and dynamic (Section 5.3) approaches.

5.1 Experimental setup

In our experimental evaluation, we used two multicore platforms with distinct characteristics and we adopted representative metrics to evaluate the results. We give more details about the multicore platforms in Section 5.1.1 and metrics in Section 5.1.2.

5.1.1 Multicore platforms

In order to conduct our experiments, we selected two distinct multicore platforms. **SMP-24**: a multicore platform based on four six-core Intel Xeon X7460. Each group of two cores shares a L2 cache (3MB) and each group of six cores shares a L3 cache (16MB). **SMP-16**: a multicore platform based on four quad-core Intel Xeon E7320 with 2MB of L2 cache shared per pair of cores. All experiments were carried out with exclusive access to these platforms. Table 5.1 summarizes the hardware and software characteristics of both platforms.

	Characteristic	SMP-24	SMP-16
Hardware	Processor	Intel Xeon X7460	Intel Xeon E7320
	Number of cores	24	16
	Number of sockets	4	4
	Clock (GHz)	2.66	2.13
	Last level cache (MB)	16 (L3)	2 (L2)
	DRAM capacity (GB)	64	64
	Name / version	**SMP-24**	**SMP-16**
Software	Linux kernel	3.2.0-2	2.6.18
	GCC	4.6.3	4.1.2
	TinySTM	1.0.3	1.0.3
	STAMP benchmarks	0.9.10	0.9.10
	Eigenbench	0.8.0	–

Table 5.1: Overview of the multicore platforms and softwares.

We selected TinySTM as the STM system for all the following experiments. As we previously explained, TinySTM is lightweight, efficient and its implementation has a modular structure that eased the inclusion of our thread mapping approach. Additionally, it can be configured to use different conflict detection and resolution policies.

Concerning the TM applications, we used STAMP benchmarks to evaluate the static approach. The reason is two fold: (i) they are realistic applications; and (ii) they can profit from static thread mapping, since most of the transactions within each application has similar behavior. For all applications, we used the largest input sizes available in the original implementation. Kmeans and vacation were configured with the low contention parameters. To evaluate the dynamic thread mapping, we used Eigenbench to create more diverse workloads.

5.1.2 Performance metrics

We use different metrics to characterize TM applications and evaluate performance: execution time, speedup, transactional time ratio, abort ratio and cache miss ratio. The speedup metric is calculated by dividing the execution time of the sequential application (*i.e.*, transactionless) by the execution time of the parallel

TM application with n threads.[1] Thus, the speedup of a parallel TM application running with n threads is given by Equation 5.1. Particularly, the execution time was measured by the `gettimeofday()` function. The execution times and speedups presented in the results represent arithmetic means of at least 30 executions.

$$Speedup(n) = \frac{Time_{sequential}}{Time_{parallel(n)}} \tag{5.1}$$

The two metrics that characterize TM applications are obtained as follows. The transactional time ratio is the percentage of the total time an application spends inside transactions. The abort ratio is calculated as presented in Equation 5.2, where $Transactions\ issued$ means the number of executed transactions (aborted + committed).

$$Abort\ ratio = \frac{Aborted\ transactions}{Transactions\ issued} \tag{5.2}$$

Finally, Equation 5.3 shows how we compute the LLC miss ratio. Particularly, $LLC\ Misses$ and $LLC\ Accesses$ mean the number of Last-Level Cache misses and Last-Level Cache accesses. They are both obtained form hardware counters through the PAPI interface [Ter+10].

$$LLC\ miss\ ratio = \frac{LLC\ Misses}{LLC\ Accesses} \tag{5.3}$$

5.2 Static thread mapping analysis

We first evaluate the accuracy of our ML-based approach for performing static thread mapping on TM applications. The evaluation is carried out as follows:

- **Input instances for the ML**: we used the STAMP applications as our input instances and TinySTM as our target STM library as explained in Section 4.3. We also varied the conflict detection (eager and lazy) and resolution (suicide and backoff) strategies to feed the learning process.

[1] Since we assign one thread per core, the number of threads is always equal to the number of cores used.

- **Prediction**: the prediction is performed at runtime after a warm-up profiling period. We observed that most of the transactions within each STAMP application have similar behavior during the whole execution. For that reason, predictions were based on the profiling of a very small sample of the first 100 committed transactions of each application at runtime without adding any important overhead. After profiling those transactions, the previously learned decision tree is traversed using the profiled information and the predicted thread mapping strategy is applied and remains unchanged during the whole execution of the application.

- **Evaluation**: we evaluate the accuracy of our ML-based approach on all STAMP applications. Since we also trained the decision trees using the STAMP applications, using the same set of applications to train and evaluate is unfair. Due to that, the decision tree used to predict the thread mapping strategy of an application is trained without the application itself. For instance, to evaluate the ML-based approach on **intruder**, we use all STAMP applications other than **intruder** as the input instances to feed the learning process. This leave-one-out cross-validation shows that our ML-based solution is able to predict thread mapping strategies for *new unobserved instances*.

The number of experiments is very large if we consider all possible combinations of parameters: 8 applications (STAMP) $\times$ 2 platforms (SMP-16 and SMP-24) $\times$ 4 STM parameters (eager, lazy and suicide, backoff) $\times$ 4 thread counts (2, 4, 8 and 16 threads). Thus, we split the analysis into two subsections. In each subsection, we consider a set of the STAMP applications and the two platforms, while fixing one of the remaining parameters (STM parameters or thread count). The complete set of results from all applications on both platforms can be found in Appendix A.

It is important to mention that our ML-based approach cannot present better performance compared to the best thread mapping strategies. In the best cases, the ML-based approach predicts a thread mapping strategy that is the best one for the application, STM configuration and thread count. In those cases, the speedup of the ML-based approach is very close to the one of best mapping. This is possible since the prediction is based on the profiling of the first 100 committed transactions,

which does not add any important overhead. On other cases, the speedup may vary depending on the performance obtained from the predicted thread mapping strategy.

In Section 5.2.1, we evaluate the ML-based approach to perform static thread mapping when varying the concurrency. Then, in Section 5.2.2, we evaluate it when varying the STM parameters. Finally, in Section 5.2.3, we draw global conclusions about the performance of the ML-based approach.

5.2.1 Varying concurrency

The first aspect we want to evaluate is how the static thread mapping based on ML performs when facing different levels of concurrency. To do so, we executed all applications from STAMP on both multicore platforms applying each one of the thread mapping strategies and our ML-based approach. This was done for different thread counts, *i.e.*, 2, 4, 8 and 16 threads. For all those experiments, we used the following STM parameters: eager conflict detection and suicide conflict resolution.

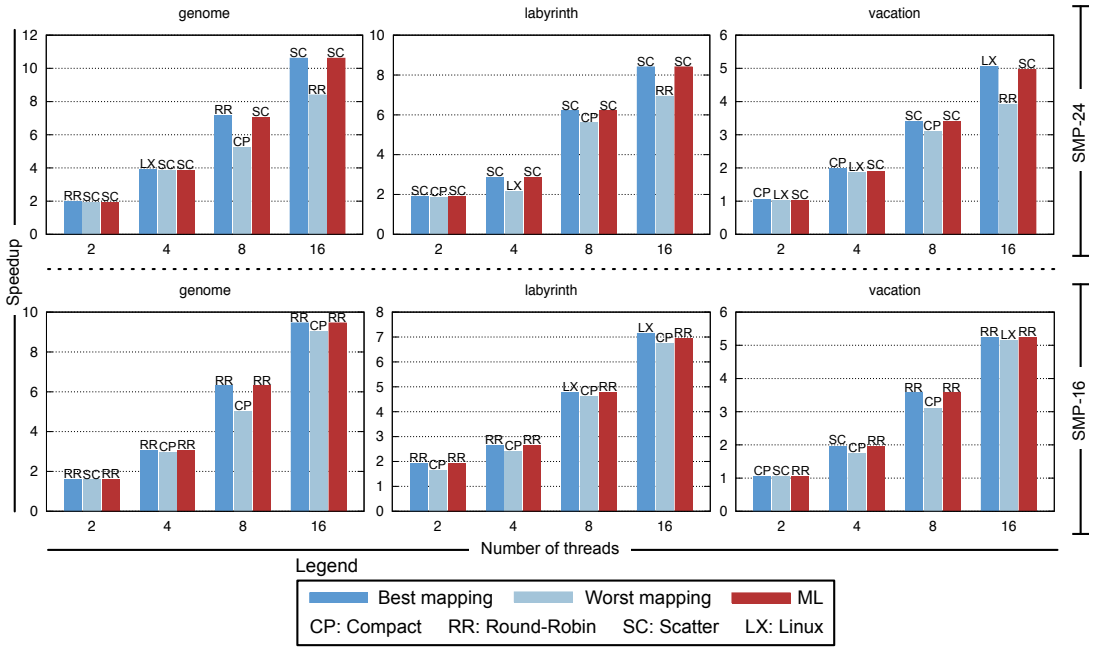


Figure 5.1: Speedups of the best and worst thread mappings in comparison to the ML when varying concurrency.

Although the STAMP is composed of 8 applications, only few scale considerably well. Because of that, we present the results of a subset of these applications that have good scalability to carry out a deeper analysis. Figure 5.1 shows the results obtained from 3 applications on both platforms. We compare the speedups obtained when applying the best, the worst and the predicted thread mapping strategies.

As it can be observed, the ML-based approach usually predicted good thread mapping strategies for these subset of applications on both platforms. We observed more important impacts on the SMP-24. This is due to the fact that this platform has a more complex cache hierarchy (two levels of shared caches). For the total of 24 different combinations of parameters in this subset of applications on two platforms (3 applications $\times$ 4 thread counts $\times$ 2 platforms), the ML-based approach selected the best thread mapping policy 14 times. For the remaining experiments, it selected a thread mapping strategy that was better then the worst strategy 7 times and it selected 3 times the worst strategy (genome with 2 and 4 threads and vacation with 2 threads). The results of all STAMP applications on both platforms are shown in Appendix A.

Genome and vacation have similar characteristics. Most of their execution time is spent inside transactions and although these transactions perform several concurrent accesses to shared data structures on memory, they present low contention (abort ratio). This indicates that transactions usually access disjoint data. These characteristics explain the results we obtained. On the one hand, there is no significant impact on changing the thread mapping strategy when the number of threads is low. On the other hand, as the number of threads increases, the contention to access the data on memory also increases. Such contention generated by several threads accessing the same cache are alleviated when thread mapping strategies such as scatter and round-robin are applied.

Labyrinth has a moderate abort ratio when it is executed with the STM parameters eager-suicide. This increases the probability of having many transactions being executed at the same time and intensifies the contention on cache memories. Compact was usually the worst thread mapping for labyrinth because it forces threads to share the same cache, increasing even more the contention. Scatter alleviates the contention on labyrinth and presented the best performance among all thread mapping strategies on SMP-24. On SMP-16, although linux was the best one for high

thread counts, it was just slightly better than scatter.

5.2.2 Modifying the STM parameters

The second aspect we want to evaluate is how the static thread mapping based on ML performs when facing different STM parameters (conflict detection and resolution policies). Similarly to the previous experiments, we executed all applications from STAMP on both multicore platforms applying each one of the thread mapping strategies and our ML-based approach. This was done for all possible combinations of STM parameters (eager-suicide, eager-backoff, lazy-suicide and lazy-backoff). For all those experiments, we fixed the number of threads to 8.

Due to the high number of experiments, we restrict our analysis to a subset of 3 applications on both platforms. The complete set of results from all applications on both platforms can be found in Appendix A. Figure 5.2 shows the results obtained from bayes, labyrinth and kmeans.

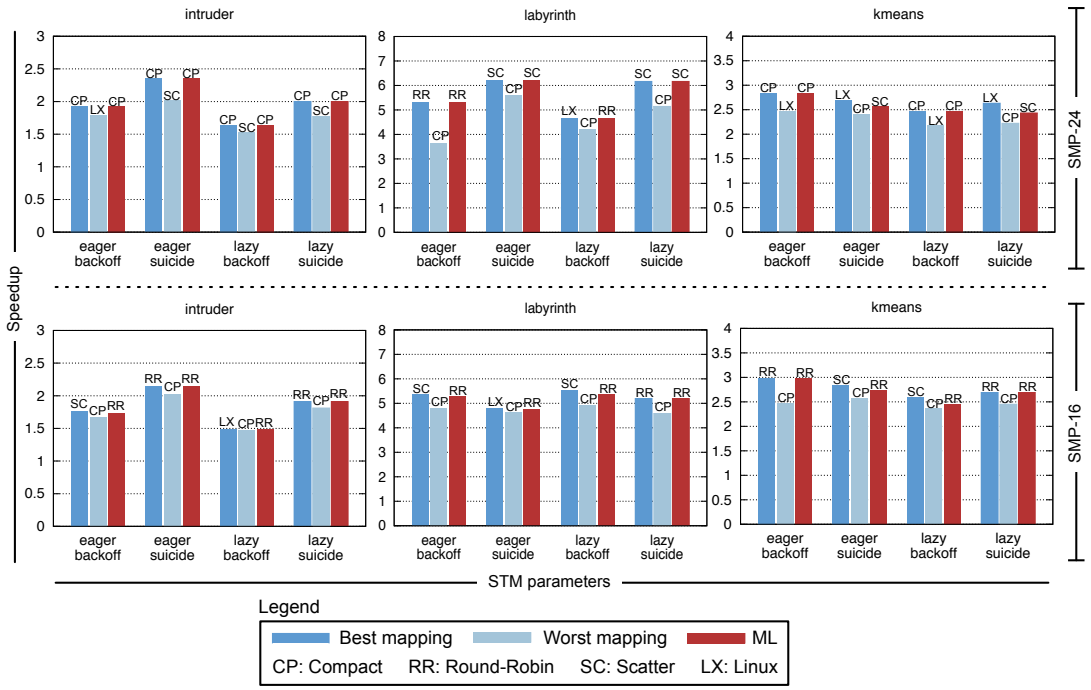


Figure 5.2: Speedups of the best and worst thread mappings in comparison to the ML when varying the STM parameters.

Intruder presents high contention regardless of the STM parameters. It executes several short transactions that usually conflict with each other. Compact can be beneficial in these cases, since threads share all levels of the cache memory and the conflicts of short transactions can be solved quickly. Our results show that compact was always the best thread mapping strategy for intruder on the SMP-24. On the contrary, although we observed very low variations on SMP-16, compact was usually the worst thread mapping strategy. It seems that the L2 caches on this platform are too small to guarantee good performances when two memory-intensive threads share the same L2.

The abort ratio on labyrinth is reduced with lazy conflict detection and backoff conflict resolution policies. In those cases, we usually observed even more performance gains when applying thread mapping strategies that avoid cache sharing such as scatter and round-robin in comparison to compact. As we previously pointed out, compact usually increases the contention on the caches on labyrinth.

Kmeans has very low abort ratio and short transactions. We observed less performance improvements with compact on the SMP-16 due to the contention on the small caches. However, on the SMP-24 we observed a different situation. Compact was the best thread mapping strategy with backoff conflict resolution. Backoff forces aborted transactions to wait for random delay (which increases exponentially with every abort) before restarting the transaction. This reduces the number of aborts as well as the cache contention on kmeans. The contrary happens with suicide, where compact presented the worst performance.

5.2.3 Overall results

In the previous sections, we show that our ML-based approach usually made good predictions for the analyzed applications. In this section, we intend to show that this approach also makes good predictions on average, when we consider all thread counts and STM configurations.

Figure 5.3 shows how far the performance of each individual thread mapping strategy and our ML-based approach are from the oracle performance. The average performance of individual thread mapping strategies as well as the performance of our ML-based approach are normalized to the average performance of the oracle.

The oracle represents the maximum performance that can be obtained, since it considers the best thread mapping strategy for each combination of parameters. A combination of parameters is an instance of an application/STM configuration/thread count/platform. In the last column, we also present the overall performance for each strategy.

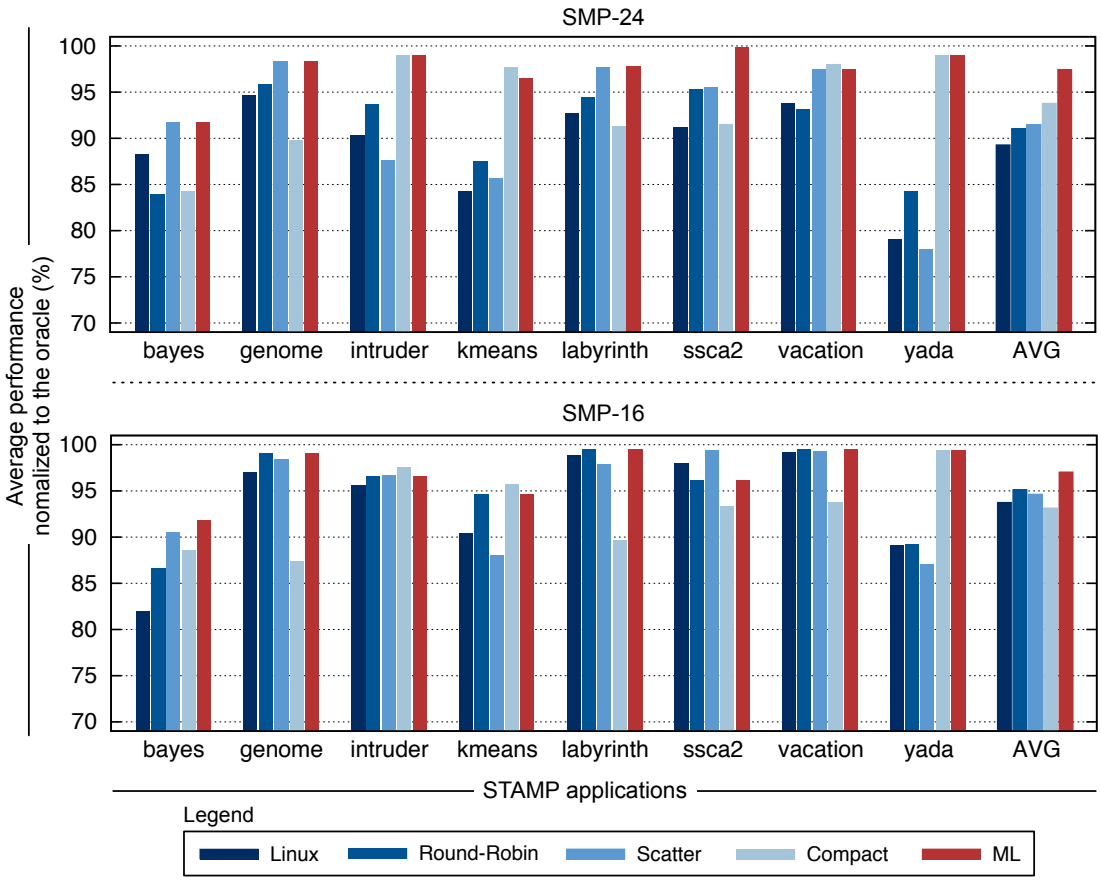


Figure 5.3: The average speedup of all benchmarks considering the fixed thread mapping strategies, our ML approach and the oracle on both platforms.

Overall, the ML-based approach presented fairly good predictions on both platforms. On the SMP-24 and SMP-16, the average performance of the ML-based approach was 97,44% and 97,04% of the oracle performance respectively. On the SMP-24, linux was the worst thread mapping strategy on average, achieving 89,27% of the oracle performance. Differently, compact was the worst one on the SMP-24,

achieving 93,14% of the oracle performance.

We also observed that some thread mapping strategies led to very poor performance in some cases. For genome on SMP-16, the compact strategy achieved 87,34% of the oracle performance. On SMP-16, on the other hand, the scatter strategy achieved 77,92% of the oracle performance. These examples confirm that choosing the wrong thread mapping strategy can lead to high performance loss and suggests that a flexible approach to select the most suitable thread mapping strategy is required.

In cases in which a single thread mapping strategy is sensitive to application/STM parameters, the ML-based approach achieves much higher average speedup compared to these single strategies. This stems from the fact that the machine learning approach considers the STM system and application features to choose a thread mapping strategy. This reduces the variability and leads to better results, as we observe in bayes on SMP-16 and ssca2 on SMP-24.

5.3 Dynamic thread mapping analysis

In this section, we present the performance evaluation of our dynamic thread mapping approach. All the following experiments were carried out on the SMP-24. This platform features a more complex memory hierarchy than SMP-16 and thus allows us to perform more interesting analyses.

In Section 5.3.1, we first define the set of workloads that will be used to compose applications with different phases. These workloads are then used throughout the following sections to evaluate our dynamic thread mapping. Contrary to the evaluation of the static thread mapping, we do not include the STAMP applications in the analysis of the dynamic thread mapping. Because of that, we used all STAMP applications during the learning phase.

The performance evaluation is carried out as follows. In Section 5.3.2, we compare the dynamic thread mapping with the static approach. In the remaining sections, we evaluate the dynamic approach in different scenarios while varying the concurrency (Section 5.3.3), STM parameters (Section 5.3.4) and the number of phases per application (Section 5.3.5). Finally, we take a closer look on how

our dynamic thread mapping reacts when it encounters several different phases (Section 5.3.6).

5.3.1 Workloads

Since most of the transactions within each STAMP application usually have very similar behavior, they are not suitable for the evaluation of our dynamic thread mapping approach. For this reason, we used EigenBench [Hon+10] to create new TM applications with different phases. EigenBench allows a thorough exploitation of the orthogonal space of TM applications characteristics.

Varying all possible orthogonal TM characteristics involves a high-dimensional search space [Hon+10]. Thus, we decided to vary 4 out of 8 orthogonal characteristics that govern the behavior of TM applications (Table 5.2).

Characteristic	Definition	Values
Tx Length	Number of shared accesses per transaction	$\downarrow$ short (≤ 64) $\uparrow$ long (≥ 128)
Contention	Probability of conflict	$\downarrow$ low-conflicting ($< 30\%$) $\uparrow$ contentious ($\geq 30\%$)
Density	Fraction of the time spent inside transactions to the total execution time	$\downarrow$ sparse ($< 80\%$) $\uparrow$ dense ($\geq 80\%$)
Concurrency	Number of concurrent threads/cores	2 – 16

Table 5.2: TM characteristics used to compose our set of workloads.

We used the first three characteristics presented in this table (*i.e.*, transaction length, contention and density) to create a set of workloads. Since we assume two possible discrete values for each one, we can create a total of 2^3 distinct workloads (named $W_1, W_2, \ldots, W_8$) by combining those values. Table 5.3 describes these workloads.

Each workload is composed of 1,000,000 transactions. For multithreaded executions of these workloads, the total amount of transactions is divided equally among the concurrent threads. For instance, on a multithreaded execution of a workload with two threads, each thread will execute 500,000 transactions. The fourth orthogonal characteristic is **concurrency** and it is further discussed in Section 5.3.3.

Characteristic	Workloads							
	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8
Tx Length	↓	↓	↓	↓	↑	↑	↑	↑
Contention	↓	↓	↑	↑	↓	↓	↑	↑
Density	↓	↑	↓	↑	↓	↑	↓	↑

Table 5.3: Transactional workloads.

5.3.2 Dynamic thread mapping vs. static thread mapping

Our first set of experiments explores the effectiveness of our dynamic thread mapping in comparison to the thread mapping strategies individually. We derived a set of applications from the 8 distinct workloads discussed in Section 5.3.1. We fixed the number of phases to 3, thus each application will be composed of three workloads. Therefore, all possible applications composed of three distinct workloads is determined by the number of k-combinations from a given set of n elements, *i.e.*, $C_k^n = C_3^8$, which results in 56 applications (named A_1,A_2,...,A_{56}). Thus, the set of applications can be represented as follows: $A_1 = \{W_1, W_2, W_3\}, A_2 = \{W_1, W_2, W_4\}, \ldots,$ $A_{56} = \{W_5, W_6, W_7\}$. Since each workload has 1,000,000 transactions, applications composed of 3 phases will have 3,000,000 transactions. Phases (workloads) are parallelized using Pthreads and there is no synchronization barrier between phases. This means that threads may not be computing the same workload at the same time.

We ran all the applications with each one of the static thread mappings (compact, scatter and round-robin), linux and our dynamic approach. Figure 5.4 presents the relative gains of our dynamic thread mapping when compared to the **best** and **worst** single thread mappings. The relative gain is given by $1 - \frac{\overline{x}_d}{\overline{x}_s}$, where $\overline{x}_d$ and $\overline{x}_s$ are mean execution times of at least 30 executions using the dynamic and the best/worst single thread mapping, respectively. Thus, positive values mean performance gains whereas negative values mean performance losses. All applications were executed with 4 threads and TinySTM was configured with lazy conflict detection and backoff conflict resolution.

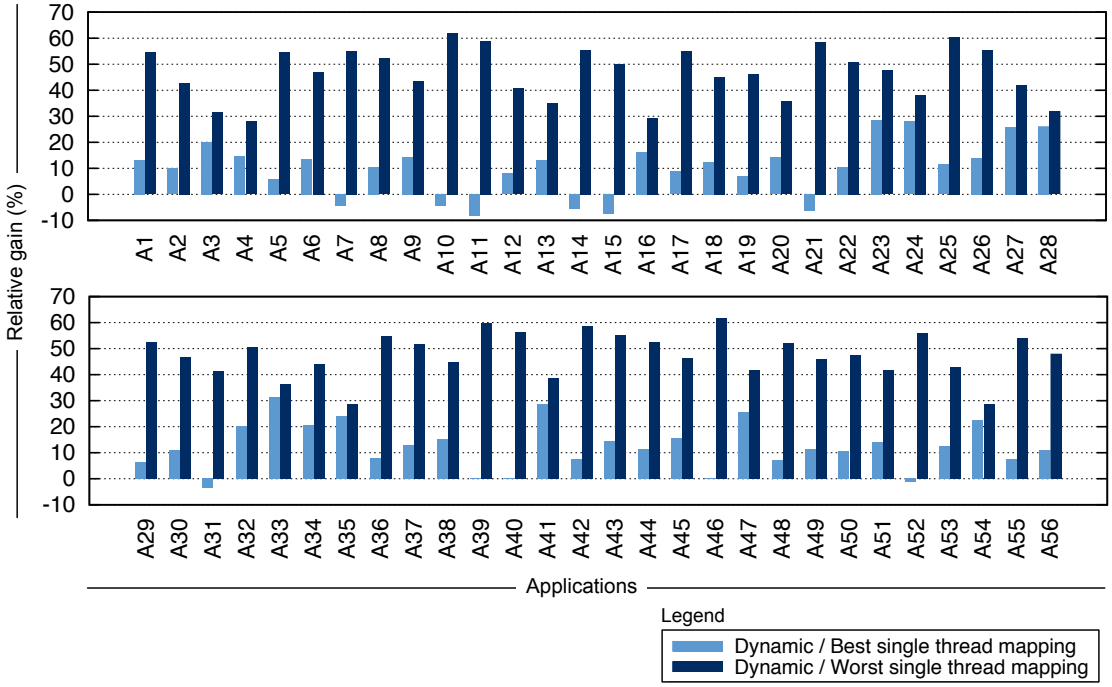


Figure 5.4: Relative gains of the dynamic thread mapping compared to the best and worst static mappings on applications composed of 3 phases (A_1 to A_{56}).

We can draw at least two important conclusions from these results. Firstly, the thread mapping strategy had an important impact on the performance. This can be easily observed when comparing the relative gains between the best and worst single thread mappings. Secondly, our dynamic thread mapping usually improved the performance of the applications by switching to an adequate thread mapping strategy in each phase. We achieved performance gains up to 31% and 62%, when comparing to the best and worst single thread mappings respectively. However, our dynamic thread mapping did not deliver performance improvements on 3 applications and presented some performance losses in 8 applications when comparing with the best single thread mapping strategy. In the case of A_{10}, A_{11} and A_{46}, a single thread mapping strategy was best for all phases (compact in these specific cases), thus we cannot expect performance improvements by using our dynamic approach. The performance losses were due to wrong decisions of the predictor, which did not select the best thread mapping strategy on all phases. The maximum performance loss was about 8% (A_{43}).

5.3.3 Varying concurrency

Our second set of experiments focuses on the performance impacts of the thread mapping strategies when varying the number of threads. We selected 4 interesting cases. Cases 1 and 2 are applications that presented a single best thread mapping strategy for all thread counts. Cases 3 and 4 are applications whose the best single thread mapping varied according to the number of threads. On all experiments, TinySTM was configured with lazy-backoff.

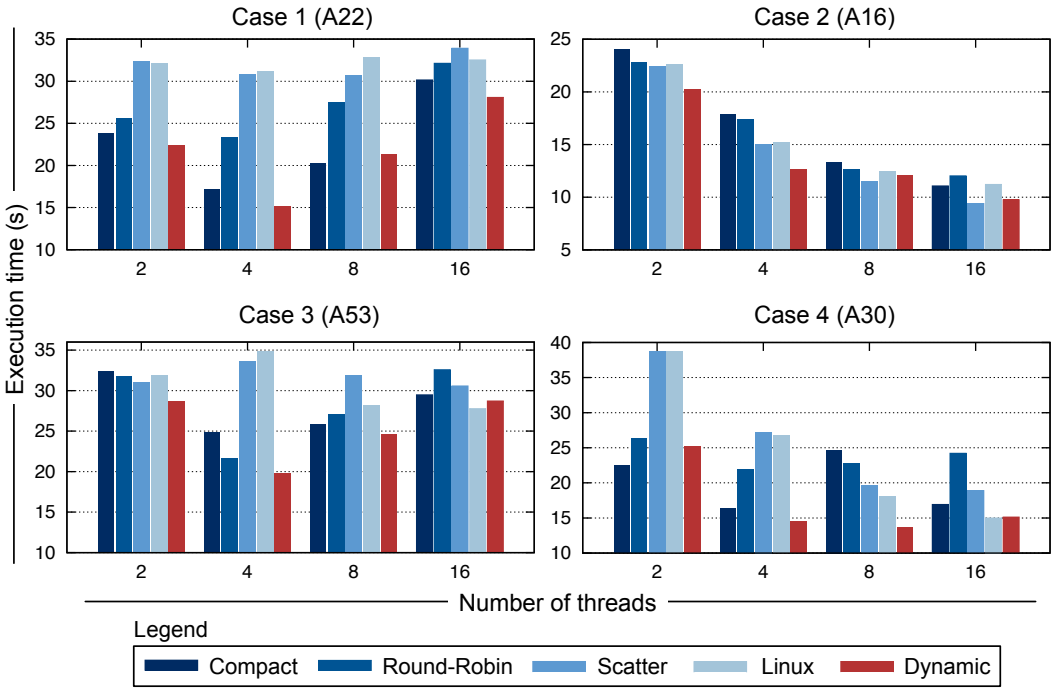


Figure 5.5: Execution times when varying the number of threads.

Figure 5.5 compares the execution times of the four single thread mapping strategies with our dynamic thread mapping mechanism. We do not consider more than 16 threads for two reasons: (i) placing threads on different cores when all available cores are used does not impact the overall performance because the applications tend to communicate uniformly, and (ii) most of our workloads did not scale beyond 16 threads.

In Case 1 (W_1, W_2, W_3), the best single thread mapping for all thread counts was

compact. The abort ratio observed in W_1 and specially in W_3 increases as the number of threads increases. Differently, W_2 presents very low abort ratio regardless of the number of threads. Since W_1 and W_3 have very low LLC miss ratios, compact presents better gains than other static thread mappings with 4 and 8 threads. In these cases, the abort ratio is moderate and threads can share the same cache without adding too much contention due to the low LLC miss ratio. Our dynamic approach selected scatter on W_2 and compact for the others when the abort ratio was low or moderate (2 and 4 threads) and round-robin when the abort ratio was high (8 and 16 threads). The use of round-robin when the abort ratio was too high reduced the contention on the same cache and thus improved the performance.

Contrary to the Case 1, scatter was the best single thread mapping for all thread counts on Case 2 (W_4, W_5, W_6). In this application, W_4 and W_6 have very low abort ratios and good scalability whereas W_5 has poor scalability and moderate to high abort ratios depending on the number of threads. The most important workload is W_6, since it is the most time-consuming. Regardless of the number of threads, scatter reduces considerably the execution time of W_4 and W_6 and thus governs the overall performance of Case 2. In most cases, our dynamic thread mapping applied scatter on these workloads. On W_5, it applied compact when the abort ratio was moderate (2 threads) and round-robin when the abort ratio was high (4, 8 and 16 threads).

Case 3 (W_4, W_6, W_7) represents a scenario in which the best single thread mapping strategy relied on the number of threads (scatter, round-robin, compact and linux with 2, 4, 8 and 16 threads respectively). Among all applications, this was the one that presented the highest sensibility to the number of threads. As a consequence, the abort ratio and LLC miss also change significantly as we change the number of threads. Overall, our dynamic approach usually presented better results than other static thread mapping strategies. We observed an exception with 16 threads, in which linux was slightly better.

Finally, in case 4 (W_1, W_2, W_8), we observed that compact was best for low thread counts whereas linux was best for high thread counts. With low thread counts, the abort ratio was moderate whereas with high thread counts, it was very high. Again, the dynamic thread mapping usually presented equivalent or better results than individual thread mapping strategies.

5.3.4 Modifying the STM parameters

Our third set of experiments focuses on the performance impacts of the thread mapping strategies when varying the conflict detection and resolution policies. For those experiments, we also selected 4 interesting cases to be analyzed in this section while fixing the number of threads to 4. The results are shown in Figure 5.6.

The eager-suicide configuration led to high abort ratios on 2 out of 3 phases in case 1 (W_2, W_4, W_8). In those phases, there were many transactions that constantly aborted, increasing the contention on the cache when the compact was used. With lazy-backoff, the abort ratio was considerably reduced and then compact presented better performance than other static thread mapping strategies. In all cases, the dynamic approach presented better performance than individual thread mapping strategies, applying compact when the abort ratio was moderate and scatter when the abort ratio was low.

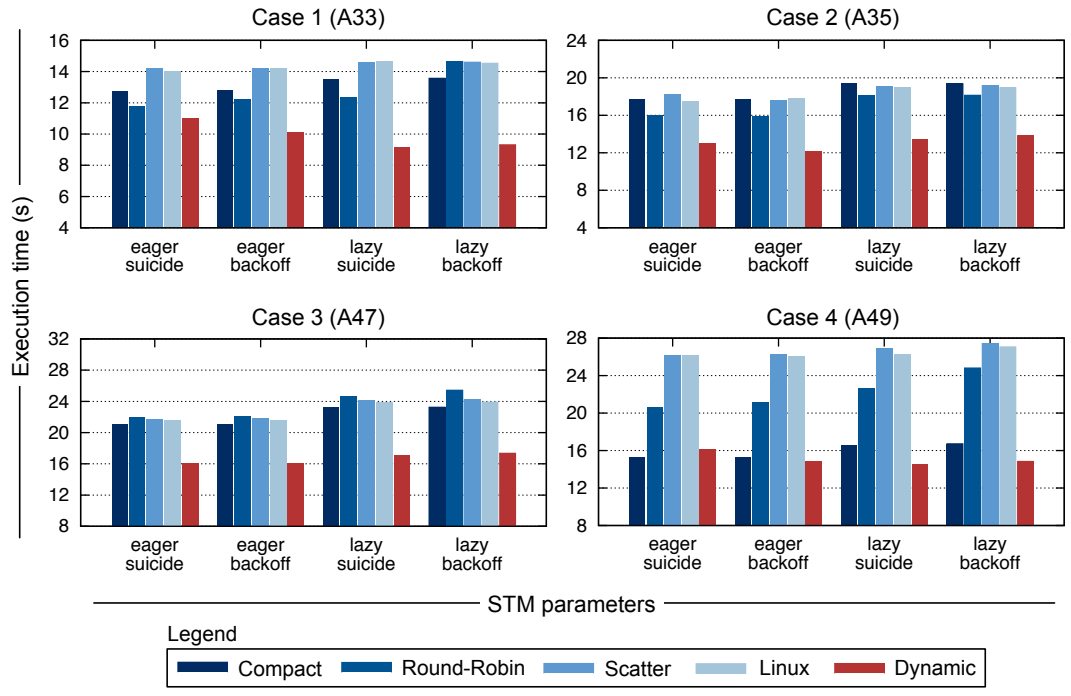


Figure 5.6: Execution times when varying the STM parameters.

In case 2 (W_2, W_6, W_8), we observed low influence of the conflict detection and resolution policies on the performance of thread mapping statics. In this application, there are two workloads with opposite characteristics: one has very low abort ratio and moderate LLC miss ratio whereas the other has moderate abort ratio and low LLC miss ratio. These two are the most time-consuming phases in this application. For the former, scatter is better because low abort ratio usually indicates transactions that access disjoint data and this strategy makes more cache available for each thread. For the latter, on the contrary, compact is better because conflicts can be solved faster without adding too much contention on the cache. Since scatter is better for one phase and compact is better for the other, these two strategies present very similar results as shown in the figure. Since our dynamic approach selected scatter for the former and compact for the latter and the third phase has little impact on the overall execution time of this application, the dynamic approach presented lower execution times than them. The best static thread mapping was round-robin because it presents a good tradeoff between these two extremes (compact and scatter).

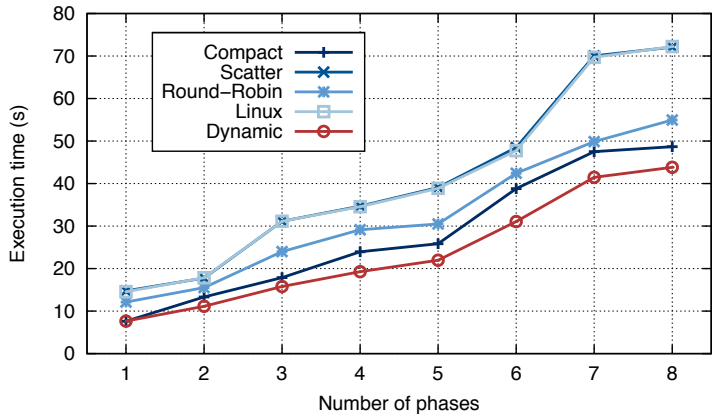
Static thread mappings seem to present low influence on the execution times in case 3 (W_1, W_4, W_6). However, we achieved significant performance gains in comparison to the static mappings because this application has a mix of very distinct workloads, with a different best thread mapping strategy for each workload. Compact presented better performance than other static thread mappings because it reduces considerably the execution time of the most time-consuming phase (W_1). Our dynamic approach selected compact in this case and scatter for the other two phases.

Finally, in case 4 (W_1, W_4, W_8), we have an application that is very similar to case 3. The difference is that the workload W_6 is substituted by W_8. The result is that compact is now the best static thread mapping, since W_1 and W_8 are the most time-consuming phases and the best thread mapping strategy for both is compact. We observed better performance gains of our dynamic approach in comparison to compact with lazy conflict detection. This is due to the fact that the number of false conflicts was reduced, resulting in much lower abort ratios on two workloads (W_4 and W_8). In those cases, our dynamic approach selected scatter, which gave better results than compact.

5.3.5 Varying the number of phases

So far, we have analyzed the performance of our dynamic thread mapping approach on a set of applications composed of 3 phases. In this section, we intend to analyze the performance of individual thread mapping strategies across the 8 workloads. To do so, we created 8 different applications by varying the number of phases in each application.

The gains obtained with the dynamic approach over the static ones are in fact directly related to the diversity of the application phases. The more distinct are the phases, the higher will be the gain if the dynamic thread mapping correctly predicts an efficiency thread mapping for each phase. The selection of the workloads to be used to compose the applications was done as follows. Firstly, we picked one of the 8 workloads to create the application with a single phase (in this case, the workload was W_5). Then, for the following applications ($Phases_{i+1}$), we added one of the remaining workloads that was very distinct from the previous application ($Phases_{i-1}$). This allows us to create applications with very distinct phases. Finally, we executed the 8 applications with 4 threads and we configured TinySTM to use lazy conflict detection and backoff conflict resolution.



Phases	Workloads
1	W_5
2	W_5, W_2
3	W_5, W_2, W_3
4	W_5, W_2, W_3, W_1
5	W_5, W_2, W_3, W_1, W_4
6	$W_5, W_2, W_3, W_1, W_4, W_6$
7	$W_5, W_2, W_3, W_1, W_4, W_6, W_7$
8	$W_5, W_2, W_3, W_1, W_4, W_6, W_7, W_8$

Figure 5.7: Performance of individual thread mapping strategies and the dynamic approach when varying the number of phases.

Figure 5.7 shows the results we obtained after varying the number of phases (left) as well as the workloads we used in each application (right). Overall, compact

was the best among all static thread mappings. The reason for that is twofold: after a deep analysis, we concluded that (i) compact was the best strategy for 4 out of 8 workloads and (ii) in some cases it was considerably better then the others. Due to the variability criteria to selected the workloads, the dynamic approach presented better performance than individual thread mappings. As we add more phases, we increase the variability (*i.e.*, there are more distinct workloads). This explains the similar execution times of the dynamic thread mapping compared to compact with few phases and also the more important gains with more phases.

5.3.6 Dynamic thread mapping in action

In order to observe how our dynamic thread mapping reacts when it encounters several different phases, we executed the application composed of 8 phases described in Figure 5.7 (that is, $W_5, W_2, W_3, W_1, W_4, W_6, W_7$ and W_8) with our dynamic thread mapping while tracing the information obtained by the *transaction profiler* at the end of each profiling period. Figure 5.8 shows the variation of the profiled metrics during the whole execution of this application.

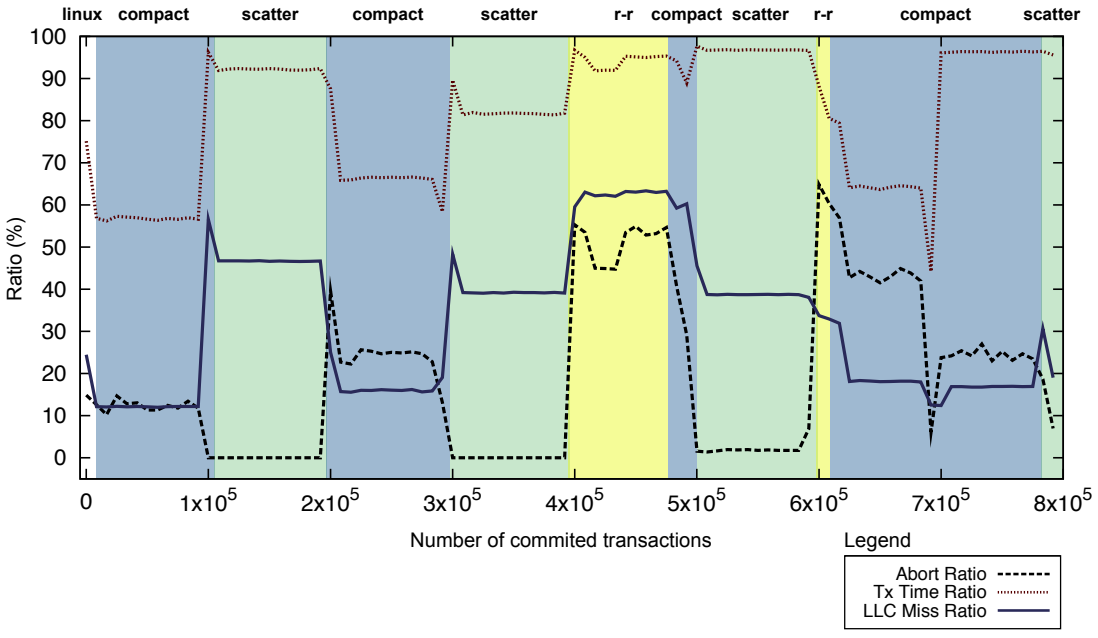


Figure 5.8: Profiled metrics during the execution of an application with 8 phases.

In this specific experiment, we executed the application with 4 threads and we set the TinySTM parameters to use lazy conflict detection and backoff conflict resolution. Vertical bars represent the intervals in which each thread mapping strategy was applied.

At the beginning, our dynamic thread mapping mechanism applies linux as its default strategy and profiles some transactions. After the first profiling period, the predictor decided to apply compact and did not switch to another strategy until it reached a different phase near 1×10^5. At this point, the predictor switched to scatter. Overall, the predictor detected more than 8 phases due to the variance of some profiled metrics but it still detected correctly the 8 main phase changes, reacting by applying a suitable thread mapping strategy for each phase. We can also observe that the variance of the profiled metrics confirms the fact that the 8 workloads have distinct characteristics.

5.4 Concluding remarks

Predicting a suitable thread mapping strategy to be applied to TM applications is not a trivial task. Due to several conflict detection and resolution mechanisms implemented by STM systems, TM applications can behave very differently on the same platform. Additionally, the target platform features have an important influence on the right choice of a thread mapping strategy. In this chapter, we intended to analyze the effectiveness of our ML-based approach in predicting suitable thread mapping strategies for TM applications.

We first evaluated the accuracy of our ML-based approach to perform static thread mapping on TM applications. For this purpose, we used all TM applications from STAMP, varying the number of threads and STM system parameters. The results showed that our ML-based approach can improve the performance of STAMP applications by up to 18.46% compared to the worst case and up to 6.37% over the Linux default thread mapping strategy. During the experiments, we applied a leave-one-out cross-validation to estimate how accurately our predictive model will perform over as-yet-unseen instances. On the SMP-24 and SMP-16, the average performance of the static approach was approximately 97% of the oracle performance.

Then, we focused on TM applications composed of more diverse workloads to evaluate our ML-based dynamic thread mapping. These workloads may go through different execution phases, each phase with potentially different transactional characteristics. In order to do so, we used EigenBench to create new TM applications with different phases. In our evaluation, we explored different configurations such as the number of threads, STM system parameters and the number of phases. Our results with dynamic thread mapping showed that there is not a single thread mapping strategy adapted for all those complex applications. Instead, we could deliver a solution capable of detecting phase changes during the execution of the applications and then predicting a suitable thread mapping strategy adapted for each phase. We achieved performance improvements up to 31% in comparison to the best static strategy.

CHAPTER 6

Related Work

We discuss in this chapter the most relevant related works concerning the main topics explored in this thesis. We separate these works in three sections. Section 6.1 presents works on the performance evaluation of TM systems and applications as well as the investigation of the behavior of TM applications. Section 6.2 presents works that make use of thread or process mapping mechanisms to improve the performance of parallel applications. Finally, Section 6.3 discusses works that use machine learning techniques to improve the performance of parallel applications.

6.1 Evaluation of TM systems and applications

Recently, some works have addressed the performance analysis of different TM systems whereas others have investigated the performance of TM applications. In this section, we highlight some of the most important works in both directions.

6.1.1 Performance evaluation of TM systems

R. Wang et al. [WLL09; Wan09] studied the performance of several STM implementations on multicore machines. Authors evaluated three STM implementations using a single benchmark (STMBench7 [GKV07]) and the Simics simulator [VIR07]. Their results showed that depending on the design choice of an STM system, the performance of STM applications can vary significantly. Although they evaluated the

performance of different STM implementations, they only focused on the influence of the memory access latencies.

C. Minh et al. [Min+08] provided descriptions and a detailed characterization of the eight STAMP applications. In particular, they measured the transaction lengths, the sizes of read and write sets, the amount of time spent in transactions and the average number of retries per transactions of all applications. Then, they used STAMP to evaluate six different TM systems. However, their performance analysis of TM systems was based on experiments performed on an execution-driven simulator. This means that the performance obtained may differ from real multicore platforms.

V. Marathe et al. [MM08], on the other hand, compared the performance of their STM solution with other STM implementations on two multicore machines. However, the performance analysis was based on four simple micro-benchmarks, which may not represent the behavior of real applications.

J. Chung et al. [Chu+06] studied 35 benchmarks from different domains. In that work, the authors translated the original synchronization mechanisms applied on all benchmarks to TM. However, they neither studied the non-trivial TM applications presented here nor they have analyzed aborts and commits, which are important metrics.

The works described above analyzed the performance of TM systems either using simulations or synthetic applications. In this thesis, we have tackled both subjects: a performance analysis of non-trivial TM applications by performing experiments on a real multicore platforms. We also analyzed the impacts of different state-of-the-art STM systems on the performance of TM applications.

6.1.2 Post-mortem analysis of TM applications

Concerning the study of the behavior of TM applications through the use of post-mortem analysis techniques, we can highlight four recent papers.

M. Ansari et al. [Ans+09] manually instrumented the Java DSTM2 STM library [HLM06] to collect execution data from TM applications. They ported three applications from STAMP suite and the Lee-TM benchmark to DSTM2 and profiled their execution. The collected information was analyzed with different metrics that bring complementary information about the behavior of TM applications. All experiments

were executed on an 8-core NUMA platform.

J. Lourenço et al. [Lou+09] proposed a monitoring framework, which collects the transactional events into a log file as well as a tool to visualize the results. Their instrumentation mechanism is based on an API, so the user must insert the API function calls within applications source codes. The API is composed of five functions to register each one of the following events: start, commit and abort of a transaction, read and write operations within transactions. Their experimental analysis was based on simple tests, *i.e.*, they used two simple data structure-based microbenchmarks (a Sorted Single Linked List and a Red Back Tree). In addition, all tests were executed with a homemade TM system derived from TL2.

F. Zyulkyarov et al. [Zyu+10] proposed a series of profiling techniques for TM applications that help developers to discover performance bottlenecks. They focused on performance problems caused by conflicting transactions, *i.e.*, the wasted work caused by aborted transactions. These techniques were implemented in a profiling framework for a specific STM system (Bartok-STM). The proposed framework is used to analyze the performance of applications from STAMP and the synthetic benchmark WormBench. However, since Bartok is a C# compiler, those TM programs had to be ported from C to C# to conform with its language-level support for TM. Indeed, this changed the behavior of some STAMP applications and the results obtained from their profilings may differ from the original STAMP applications.

D. Porter et al. [PW10] explored a different technique to characterize and tune the performance of TM applications. They proposed a formal model of TM performance and a tool based on the model called Syncchar. The Syncchar model samples the sets of addresses read and written during critical sections of a lock-based parallel application. This information is used to construct a model of the program execution that can predict the performance of the application if it used transactions. In addition to predicting the performance of a TM application, the model can be used to tune the performance, since it provides two useful metrics: data independency and conflict density. The data independency determines if transactions usually operate on disjoint or same data. The conflict density is a measure of how long the serial schedule resulting from a conflict is likely to be. Those metrics can give the developer an indication of how much parallelism they can expect from the application.

Unlike the previously cited proposals, our solution used an interception approach

based on the Linux dynamic linking mechanism. This allows us to develop a solution that does not depend on the TM application specifics. Additionally, since we do not need to change neither the TM application nor the STM library source codes, it can be easily applied with different STM systems with very few modifications. It can also be easily adapted to conform with the user needs.

6.2 Thread and process mapping

Thread or process mapping mechanisms have always the same objective, that is, to improve performance. Although the focus may be different depending on the programming model (*e.g.*, OpenMP and MPI applications), they are relevant to this thesis because they usually propose heuristics to map threads or processes on multicore platforms.

H. Chen et al. [Che+06] proposed a profile-guided approach to automatically find an optimized mapping for arbitrary message passing applications. Their approach does not inquire users' knowledge on either applications or target systems. It is composed of two steps. Firstly, they collect the communication profile of the MPI application and the network topology of the underlying cluster platform. Then, such information is used as the input of a graph mapping algorithm, which maps the communication graph of parallel applications to the system topology graph to minimize the cost of point-to-point communications. They evaluated the performance of this approach with the NASA Parallel Benchmarks (NPB) [Bai+95] and three other applications in different cluster configurations. They show that the optimized process placement generated by their algorithm can achieve significant performance improvements in comparison to the MPI default placement.

J. Zhang et al. [Zha+09] extended the approach proposed in [Che+06] to consider MPI collective communications. Basically, they proposed to transform collective communications into a series of point-to-point communication operations according to the implementation of collective communications in communication libraries. Then, they used existing approaches to find optimized mapping schemes which are optimized for both point-to-point and collective communications. The authors evaluated the performance of this approach with microbenchmarks which

include all MPI collective communications, NPB suite [Bai+95] and three other applications.

S. Hong et al. [Hon+09] proposed a dynamic thread mapping strategy for regular data parallel applications implemented with OpenMP. The applications considered in this wok consist of loop based computations manipulating arrays. The proposed strategy has two alternating phases, namely detection phase and stable phase that are separated by a remapping step. In the detection phase, they compute the number of processor cycles taken to execute each thread as well as the cache accesses. Threads are then ranked in decreasing order of their load considering these two metrics. Then, in the remapping step, threads are mapped in such a way that the load is distributed among the processors. The new mapping remains unchanged during a fixed number of iterations called the stable phase. The detection and stable phases can be repeated multiple times for the same loop to accurately capture the loop behavior at runtime, and to better adapt application execution. The efficiency of the proposed thread mapping strategy was evaluated using the Simics [VIR07] simulator and NPB suite [Bai+95] and average improvement obtained was about 13%.

M. Diener et al. [Die+10] examined data sharing patterns between threads in different workloads and used those patterns to map processes. These algorithms relied on memory traces extracted from benchmarks to find data sharing patterns between threads. During the profiling phase, these patterns were extracted by using Simics [VIR07] simulator, running the workloads in a simulated UltraSPARC machine. The proposed approach was compared to *compact*, *scatter* and the operating system process mapping strategies. The authors achieved moderate improvements in the common case and considerable improvements in some cases, reducing execution time by up to 45%.

Similarly, *E. Cruz et al.* [Cru+11] used memory traces to perform thread mapping on OpenMP applications. They proposed and evaluated a technique of process mapping that binds threads of a given application on cores and allocate their data on DRAM memories, reducing the overhead of communication among the threads that share data. This method used different metrics and an heuristic to obtain the mapping. Their results showed performance gains of up to 75% compared to the Linux scheduler and memory allocator.

In contrast to these works, we proposed a ML-based approach for deciding an

appropriate thread mapping strategy considering both application and platform features. We targeted TM applications, which tend to present a more dynamic behavior than those suited to OpenMP. We do not rely on simulations to gather information about the application, STM system and platform. Instead, we use hardware counters and software libraries to gather information about the platform and applications at runtime.

6.3 Machine learning

Machine Learning has been extensively used as a predictive mechanism to solve a wide range of problems. In this section, we briefly describe some works that rely on Machine Learning techniques to improve the performance of parallel programs.

D. Grewe et al. [GO11] proposed a ML-based compiler model that accurately predicts the best partitioning of data-parallel OpenCL tasks. Static analysis is used to extract code features from OpenCL programs. These features are used to feed a ML algorithm which is responsible for predicting the best task partitioning. The prediction is composed by two stages. In the first stage, the model predicts whether tasks should be only mapped to GPUs or CPUs. If the first stage of prediction does not lead to a conclusion, the program is then passed to the second stage which is responsible for mapping tasks to both GPUs and CPUs. The number of tasks to be mapped to GPUs and CPUs is determined by the predictor. The proposed model achieved a speedup of 1.57 over a state-of-the-art dynamic runtime approach. Additionally, it achieved speedups of 3.02 and 1.55 over a purely multicore and GPU approaches respectively.

G. Tournavitis et al. [Tou+09] proposed a two-staged parallelization approach combining profiling-driven parallelism detection and ML-based mapping to generate OpenMP annotated parallel programs. In this method, first they used profiling data to extract control and data dependencies to identify portions of code that can be parallelized. Afterwards, they applied a previously trained ML-based prediction mechanism to each parallel loop candidate in order to select a scheduling policy from the four options implemented by OpenMP (*cyclic*, *dynamic*, *guided* or *static*). They evaluated the proposed parallelization strategy against the NAS and SPEC OMP benchmarks on two different multicore platforms. The results showed that their

approach achieves 96% of the performance of the hand-tuned OpenMP NAS and SPEC parallel benchmarks.

Z. Wang et al. [WO09] proposed a compiler-based approach to map OpenMP programs to multicore processors. Particularly, it focuses on determining the best number of threads for a parallel program and how the parallelism should be scheduled. They proposed a ML-based predictor that automatically builds a model of the behavior of the machine based on prior training data. This model predicts the performance of particular mappings and is used to select the best one. In order to evaluate the performance of the proposed solution, the authors compared their approach to the default OpenMP runtime on two different multicore platforms. The results showed that the ML-based approach has better performance than the OpenMP runtime scheme.

Q. Wang et al. [Wan+11] introduced predictive mechanisms based on Machine Learning that can select an STM algorithm adapted to the TM application workload at runtime. They used two datasets as input for the off-line training. The first dataset is the result of the characterization obtained from runtime profiling: they profiled several microbenchmarks in single-threaded mode to collect runtime information such as read-only ratio, transactional and nontransacitonal work and shared memory accesses. The second dataset comes from throughput measurements of the microbenchmarks at many thread levels using each one of the 15 STM algorithms available in the RSTM system. Both datasets are used to construct adaptive policies, one based on Case-Based Reasoning (named CBS) and other based on Neural Networks (named NEAT). The authors tested both adaptive policies with all applications from the STAMP suite. At runtime, some transactions are profiled and the profiled data is used to feed the adaptive policy to choose the STM algorithm. Since most of the transactions within each STAMP application have similar behavior, applications can be profiled several times at runtime with a very low overhead (a profile of one single transaction in each profiling interval sufficed). The results showed that the proposed adaptive policies present performance improvements very close to the oracle that always chooses the best algorithm.

In contrast to these works, we are interested on the impact of thread mapping strategies on the performance of TM applications, which can be more sensitive to thread mapping due to their complex memory access patterns. Our ML-based

approach considers the TM application, STM system and platform features for predicting an appropriate thread mapping strategy. It does not require any modification in the operating system or application and it does not rely on the user's knowledge. Additionally, it is portable across different operating systems and platforms.

Although our approach and the work presented in [Wan+11] share the idea of using Machine Learning techniques to increase the performance of TM applications, they differ in several aspects. Perhaps the main difference between them comes from the use of Machine Learning for different goals: prediction of thread mapping strategies vs. selection of STM algorithms. In this sense, our approach is complementary to that work. In addition to that, they differ in terms of Machine Learning algorithms used for prediction: we use Decision Tree Learning whereas they use Case-Based Reasoning and Neural Networks. The collected information during the learning phase and runtime profiling also differs: we do not need to collect information in single-threaded mode. This would add a considerable overhead in more dynamic applications, since it would be necessary to profile several transactions in single-threaded mode to capture the workload behavior. Finally, they did not consider any information from the platform, which in some cases may play an important role.

CHAPTER 7

Conclusion and Perspectives

PROCESSOR manufacturers have responded to the demand for more processing power by delivering faster processor speeds. However, the need to achieve higher performance without driving up power consumption and heat recently became a critical concern. To respond to this problem, manufactures are now investing on multicore platforms, based on the fact that a good overall processing performance can be achieved by reducing individual core clock speeds while increasing the number of cores.

In order to develop parallel applications for those platforms, developers rely on high-level environments such as OpenMP, MPI [Qui08] and Charm [KB12]. Although these environments aim at simplifying the development of correct parallel applications, they hide several issues which impact the performance of the applications. Consequently, developers must also take care of several aspects, ranging from the platform peculiarities to the application level, if they want to obtain efficient parallel applications.

In this context, Software Transactional Memory (STM) appears as another programmer friendly alternative approach to leverage the development of parallel applications on those modern multicore platforms. It allows programmers to write parallel code as transactions, which are guaranteed to execute atomically and in isolation regardless of eventual data races. At runtime, transactions are executed speculatively and the STM runtime system continuously keeps track of concurrent

accesses and detects conflicts. Conflicts are then solved by re-executing conflicting transactions. However, as the other previously mentioned high-level environments, STM hides important aspects that may impact the performance of Transactional Memory (TM) applications. In addition to that, TM applications can behave differently depending on the characteristics of the underlying STM system.

The contributions of this thesis were situated around the analysis and improvement of the performance of TM applications on multicore platforms using low-intrusive techniques to both TM applications and STM systems. This was done without incurring any modifications to neither TM applications nor the core of the STM systems.

7.1 Contributions

The first part was dedicated to the performance analysis of TM applications (Chapter 3). First, we took a deeper look on the impacts of STM systems on the performance of TM applications. We showed that the performance of applications that use TM-based synchronization systems depend on both the applications themselves and the STM system specifics. However, STM hides important aspects that may impact the performance of TM applications and then understanding the performances achieved is a daunting task. In addition to that, the high variability of TM applications and STM systems makes it even more difficult to establish a simple yet generic solution that allows developers to accomplish this task.

We argued that developers can better understand the performances obtained from TM applications by tracing runtime events that are relevant in the context of TM applications. Thus, we proposed and implemented a prototype library called `libTraceSTM`, which can be used with different TM applications and STM systems without any changes in their original source codes. The traced data can then be used to perform runtime or post-mortem analyses. Since we store events along with their corresponding timestamps, we can obtain temporal information from events generated by TM applications. For instance, it is possible to discover if the TM application presents points of high contention (“hot spots”) or if the contention is spread out over the whole execution. Finally, we applied our tracing mechanism on three TM applications from STAMP and analyzed the obtained results. Our study

was based on the comparison of the traced data from those applications on two state-of-the-art STM systems. We showed that the traced information could help us to better understand the obtained performances.

The second part addressed the performance improvement of TM applications on multicores (Chapter 4). We observed that the performance of TM applications can be improved if we match their characteristics (along with the characteristics of the STM system) to the underlying multicore platform. More precisely, we were interested in gathering useful information from the TM applications and STM systems and then use such information to better exploit the memory hierarchy of modern multicore platforms. To deal with the large diversity of TM applications, STM system configurations and multicore platforms, we proposed to use Machine Learning to automatically predict suitable thread mapping strategies for TM applications. Our approach was based on a prior learning phase, in which we profiled several TM applications running on different STM systems to know their characteristics. Then, such information was used to feed a ML algorithm, outputting a predictor. Once this learning phase was finished, we were able to use the predictor to infer suitable thread mapping strategies to be applied to new unknown TM applications.

We implemented two approaches to perform thread mapping for TM applications on TinySTM (one of the state-of-the-art STM systems). The first approach performs static thread mapping, in which we profile TM applications during an initial warm-up period and the predicted thread mapping strategy is applied once, remaining unchanged during the whole execution. The second approach performs dynamic thread mapping, in which we profile TM applications several times during the execution, switching the thread mapping strategy to a more adequate one when necessary.

We finally evaluated the performance of our Machine Learning-based approach to perform static and dynamic thread mapping on TM applications (Chapter 5). We concluded that the static approach was fairly accurate. Considering all STAMP applications, different STM parameters and thread counts, and two different SMP platforms, our solution had on average 97% of the optimal performance provided by the oracle. Concerning the dynamic approach, we showed that it could detect different phase changes during the execution of TM applications composed of more diverse workloads, predicting a suitable thread mapping strategy adapted for each

phase. We achieved performance improvements up to 31% in comparison to the best static strategy. Overall, the average performance improvement of the dynamic approach compared to the best static strategy on each application was approximately 14% (48 out of 56 applications presented better performance with the dynamic approach) whereas the average performance loss was approximately 4% (8 out of 56 applications presented worse performance with the dynamic approach).

7.2 Future works

This research can be extended in several directions. Below, we highlight some of those possibilities:

- **Explore new architectures, systems and TM applications**. We have constantly seen efforts for a wider adoption of Transactional Memory. Recent efforts concern the addition of TM support on the latest version of the GNU Compiler Collection (GCC 4.7) and the new processors that add hardware support for TM: BlueGene/Q from IBM and Intel Transactional Synchronization Extensions (TSX) for the future multicore processor code-named "Haswell". We believe that those efforts will broaden the audience of TM and will also significantly contribute to the appearance of new real-world TM applications. In addition to that, it is expected that those new architectures will apply the NUMA design. Thus, a medium- to long-term research possibility will be to apply our ML-based predictor on those new architectures and applications. We believe that it can be possible to extend our approach to be used with NUMA platforms. However, the way data is allocated/distributed among the memory banks also influences the overall performance of applications on those platforms. For that reason, it would be necessary to consider not only the thread mapping strategies but also memory allocation policies to make better use of the NUMA memory banks. Our previous works on the impact of memory allocation policies on NUMA paltforms [Cas+09; Rib+09b; Rib+12] can be used as a good starting point.

- **Create new predictors based on other ML algorithms**. Our ML-based approach to perform thread mapping relied on a specific Decision Tree Learning

algorithm (ID3) to construct the predictor. In our approach, we used profiled data from several TM applications and STM systems as an input for the ID3 learning algorithm. Once this previous training phase was finished, we used the same predictor for all new unknown TM applications. This process is known as *off-line learning*. Another direction for future works can be the use of *online learning algorithms*. With online learning, the profiled information derived from new TM applications can be used to automatically adapt the predictor. We believe that such approach can be useful for improving the predictor accuracy on a wider range of TM applications.

- **Extend the ML-based predictor to consider other system metrics**. In this thesis, we used a set of metrics to be profiled, such as the abort ratio, LLC miss ratio and conflict detection and resolution policies. Probably, the most obvious improvement would be to consider a broader range of STM conflict detection and resolution policies. Another interesting metric that could be added is energy consumption. Energy is increasingly becoming one of the most expensive resources and the most important cost item for running a large supercomputing facility. Thus, we could imagine a situation in which we would want to use energy consumption instead of execution time as our performance metric in the ML-based predictor. In this case, the predictor would select a thread mapping strategy that consumes less energy for a certain TM application/STM system/platform configuration.

APPENDIX A

Static Thread Mapping Results

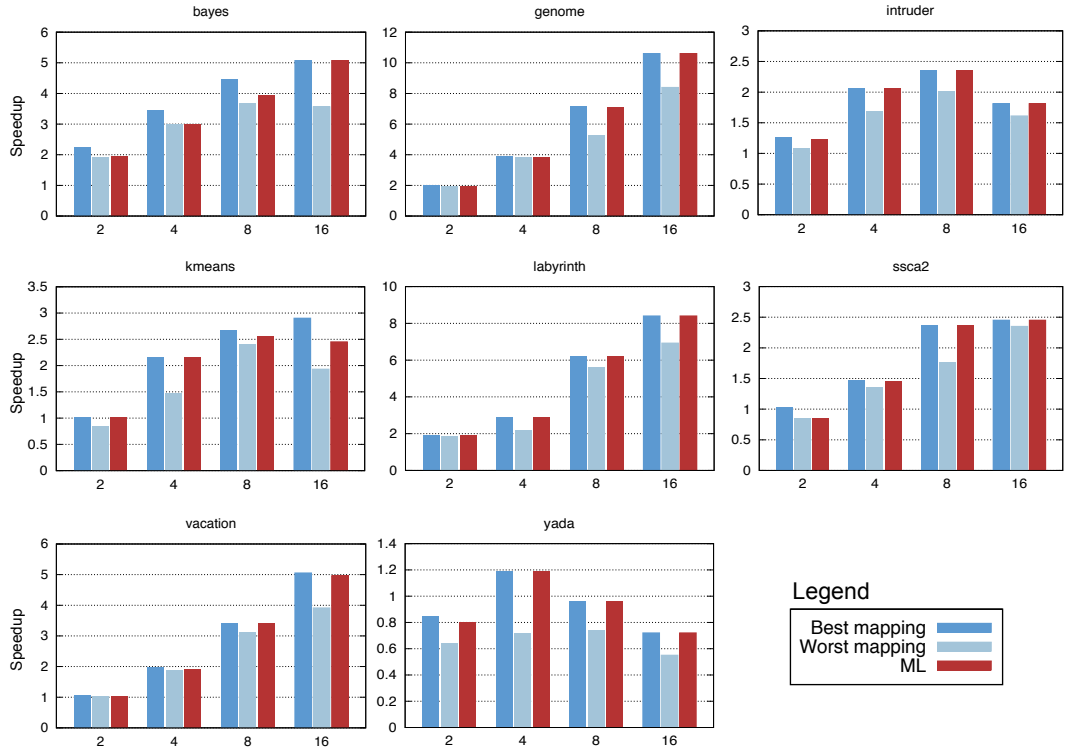


Figure A.1: Speedups of the best and worst thread mappings in comparison to the ML when varying concurrency on the SMP-24.

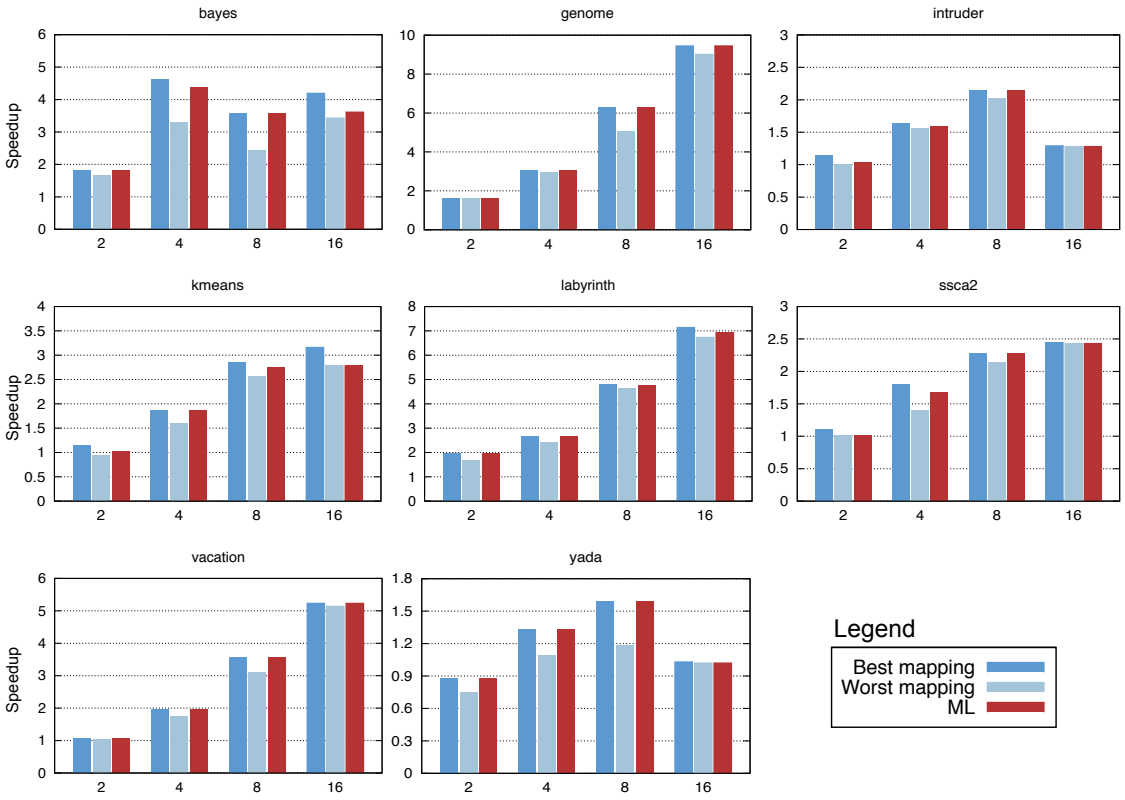


Figure A.2: Speedups of the best and worst thread mappings in comparison to the ML when varying concurrency on the SMP-16.

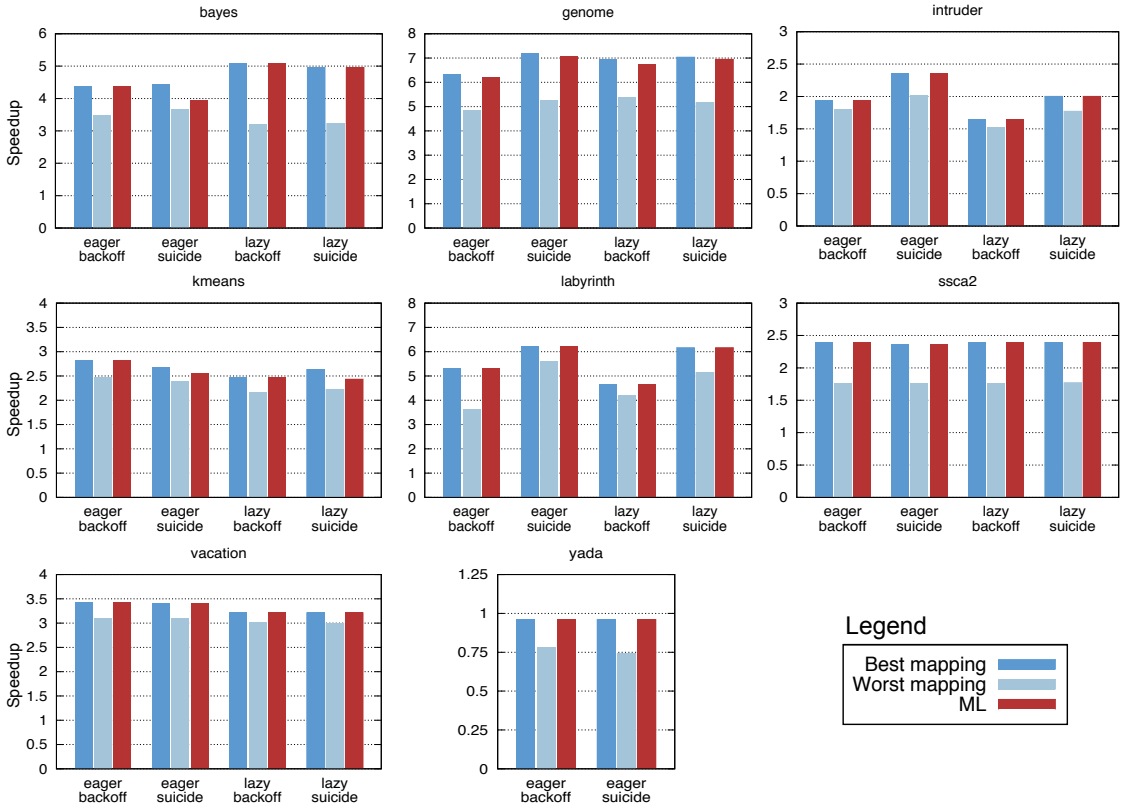


Figure A.3: Speedups of the best and worst thread mappings in comparison to the ML when varying the STM parameters on the SMP-24.

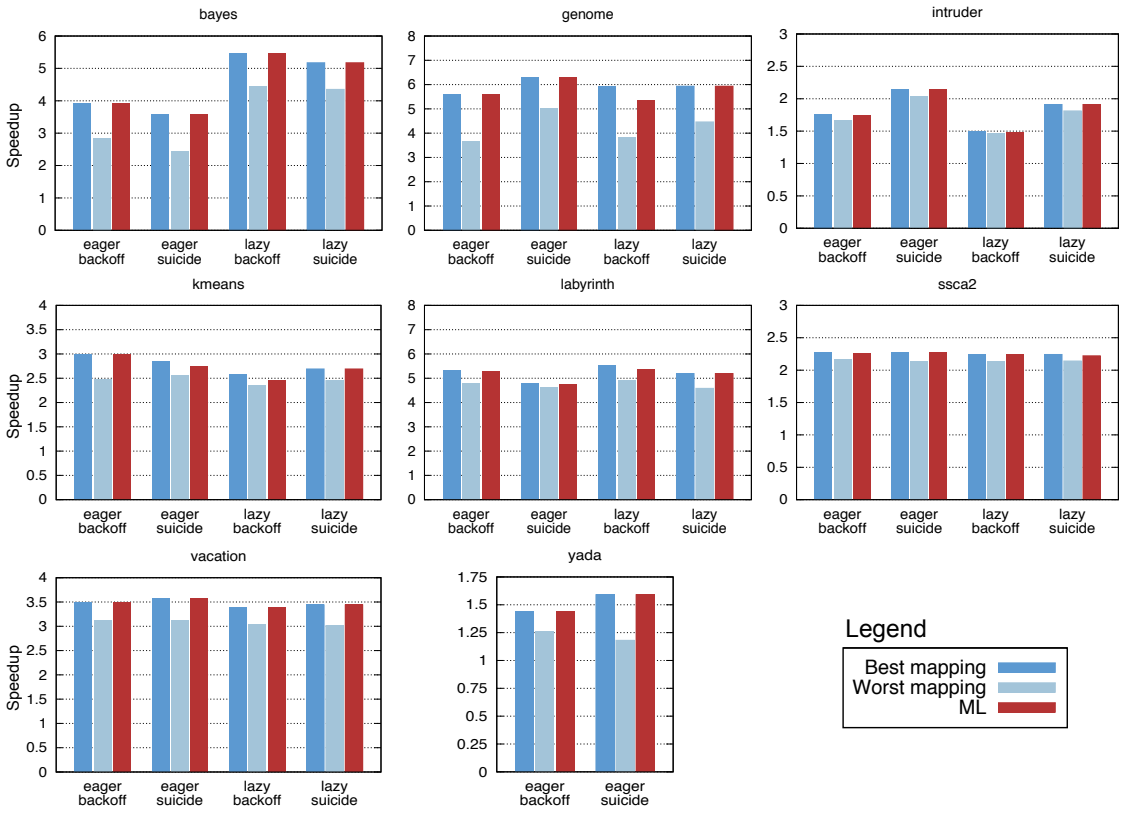


Figure A.4: Speedups of the best and worst thread mappings in comparison to the ML when varying the STM parameters on the SMP-16.

APPENDIX B

Extended Abstract in French

Optimisation de la Performance des Applications de Mémoire Transactionnelle sur des Plates-formes Multicœurs : Une Approche Basée sur L'Apprentissage Automatique

Márcio Bastos Castro

Table des matières

CHAPITRE 1

Introduction

Depuis les années quatre-vingt, l'évolution des technologies des semi-conducteurs et des architectures informatiques a permis d'améliorer significativement les performances des processeurs avec des accroissements de performances de l'ordre de 40 à 50% [LK08]. Toutefois, avec des circuits de plus en plus denses, cela a soulevé d'autres problèmes liés notamment à la dissipation de la chaleur dans des transistors. Cela a conduit les industriels à limiter la fréquence des processeurs et à opter pour des architectures multiprocesseurs et/ou multicœurs dont la dissipation thermique est limitée.

Les architectures multicœurs sont au coeur des environnements de haute performance [Asa+09]. Mais, même si elles permettent de répondre convenablement au problème de dissipation, exploiter efficacement la puissance des architectures multicœurs reste un défi majeur. Par exemple, la puissance de traitement est souvent limitée par le débit de transfert de données entre la mémoire principale et les processeurs. Ce problème est connu dans la littérature sous le nom de *memory wall problem* [McK04]. Pour répondre à ce problème, ces plates-formes disposent généralement de hiérarchies de mémoire complexes composées de différents niveaux de *cache* pour maximiser l'utilisation des données issues de la mémoire principale.

En considérant que la technologie de semi-conducteurs est capable de doubler le nombre de transistors sur une puce tous les deux ans, il est clair que le nombre de cœurs sur une puce continuera à augmenter. Par exemple, le programme de

recherche informatique mené par Intel a créé un prototype de puce contenant 80 coeurs. Dans ce contexte des architectures *many-cores*, les chercheurs explorent également l'utilisation de la technologie 3D pour fournir des mémoires *cache* à la fois larges et à faible latence empilées au-dessus des processeurs. Ce qui va permettre de réduire la consommation énergétique et d'augmenter la bande passante [HBK06].

Par ailleurs, les applications doivent évoluer pour exploiter efficacement la puissance des plates-formes multicœurs. Les applications séquentielles doivent être découpées en processus ou tâches parallèles. Chaque processus, généralement de type processus léger ou *thread* en anglais, est assigné à un processeur spécifique de la machine d'exécution. Les différents *threads* se partagent les données de l'application, nécessitant d'intégrer des mécanismes de synchronisation pour assurer la cohérence des exécutions. Cette synchronisation complexifie la programmation et induit en général un surcoût sur les performances l'application.

Les mécanismes de synchronisation traditionnelle, tels que les verrous, les *mutexes* et les sémaphores ont été largement utilisés pour la synchronisation de *threads* sur des plates-formes multicœurs. Toutefois, ils présentent des inconvénients importants. Ils sont considérés comme "mécanismes de bas niveau", puisqu'on doit contrôler explicitement l'accès aux variables partagées. Aussi, ils provoquent des blocages, donc les *threads* doivent se bloquer sur un verrou (ou un ensemble de verrous) avant de poursuivre leur exécution. Notons également que l'utilisation incorrecte de ces mécanismes peut facilement conduire à des interblocages (*deadlocks*) ou des situations du type *livelock* [Tai94]. Enfin, la synchronization peut ralentir considérablement l'évolution de l'exécution des programmes parallèles et rajoute une complexité importante dans le code source.

En raison des problèmes abordés ci-dessus, les chercheurs tentent de trouver d'autres mécanismes alternatifs. Un de ces mécanismes qui a fait l'objet d'actives recherches ces dernières années est la Mémoire Transactionnelle (*Transactional Memory* – TM). Ce modèle de programmation permet d'écrire des portions parallèles de code dans des transactions. Ces transactions sont exécutées de façon atomique et isolée les unes des autres, tout en garantissant l'exactitude de l'exécution parallèle [Dal+10 ; HLR10]. Lors de l'exécution, les opérations sont exécutées d'une façon spéculative et le système d'exécution de la TM gère les accès concurrents et détecte les conflits. Les conflits sont alors résolus en réexécutant les transactions en conflit. Ce qui évite au

programmeur de gérer explicitement la synchronisation des accès concurrents aux données partagées.

Différentes implémentations de systèmes de TM font des compromis entre les performances et la facilité de programmation. Les choix de conception les plus courants sont la Mémoire Transactionnelle Logicielle (*Software Transactional Memory* – STM), la Mémoire Transactionnelle Matérielle (*Hardware Transactional Memory* – HTM) et Mémoire Transactionnelle Hybride (*Hybrid Transactional Memory* – HyTM). La STM est une approche logicielle et ne nécessite donc pas de matériel spécifique [DGK09 ; DSS06 ; FFR08]. Au contraire, toutes les fonctionnalités de la HTM sont mises en œuvre au niveau matériel [McD+05 ; Moo+06]. La HyTM, quand à elle, est une approche hybride dans laquelle le matériel sert simplement à optimiser les performances des transactions qui sont contrôlées par une solution logicielle [Kum+06 ; Shr+06].

La STM a un certain nombre d'avantages par rapport aux deux autres approches. Elle portable sur un grand nombre de systèmes et plates-formes et offre une flexibilité dans la mise en œuvre des mécanismes et politiques de détection/résolution de conflits. En outre, des solutions matérielles et hybrides ne sont pas encore disponibles dans les processeurs commerciaux. L'inconvénient est que la STM est moins performante que les HTM et HyTM.

En général, l'efficacité des applications parallèles repose sur l'appariement du comportement de l'application avec le système et architecture sous-jacents. Plus précisément, cette question devient beaucoup plus complexe dans les STM. Il y a deux raisons à cela. La première raison est liée au modèle de la TM : comme la TM utilise la spéculation, les applications se synchronisant avec de la TM présentent des comportements irréguliers (les dépendances de données entre les *threads* ne sont connues que lors de l'exécution). La deuxième raison est liée à la STM : chaque système de STM met en œuvre ses propres mécanismes pour détecter et résoudre des conflits. Ansi, une application se synchronisant à l'aide de la STM peut se comporter différemment lorsque le système de STM sous-jacent est différent.

La Figure 1.1 illustre ce scénario suivant trois axes. Sur l'axe de la STM, nous avons presenté quelques systèmes de STM actuellement disponibles. Chacun a ses spécificités, en ce qui concerne par exemple les mécanismes pour détecter et résoudre les conflits. Sur l'axe des applications utilisant la TM, nous avons inclus les

applications et des *benchmarks* qui sont utilisés pour évaluer les systèmes de STM. Quelques *benchmarks* comme par exemple le *Stanford Transactional Applications for Multi-Processing* (STAMP) sont composés de plusieurs applications [Min+08]. Les différences entre ces applications se situent sur le niveau de la concurrence, la probabilité de conflits, la taille et le temps passé à l'intérieur des transactions, etc. Enfin, sur l'axe des plates-formes multicœurs, nous avons représenté les différences en termes de hiérarchie mémoire : plates-formes ayant uniquement des *caches* privés, uniquement des *caches* partagés ou les deux. Tous ces aspects peuvent avoir un impact important sur les performances des applications.

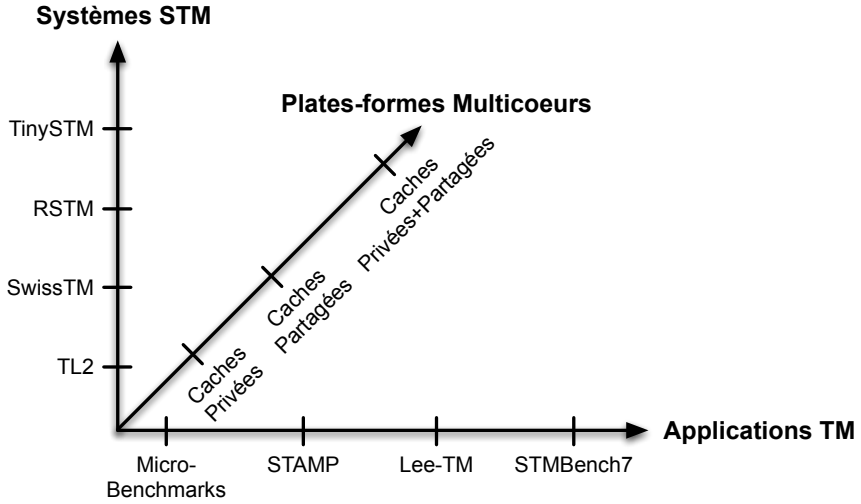


FIGURE 1.1 – Systèmes de STM, applications et plates-formes multicœurs.

Dans cette thèse, nous voulons **comprendre** et **optimiser** les performances des applications se synchronisant à l'aide de la STM sur des plates-formes multicœurs. Toutefois, cela n'est pas trivial en raison du nombre de combinaisons possibles des aspects précédemment cités. Notre objectif est donc d'analyser les performances et de proposer des améliorations en utilisant des techniques à faible intrusivité à la fois au niveau des applications et des systèmes de STM.

1.1 Contributions

La première contribution de cette thèse se situe sur l'analyse et compréhension des applications se synchronisant à l'aide de la mémoire transactionnelle sur des plates-formes multicœurs. Tout d'abord, nous étudions l'impact des systèmes de STM sur les performances des applications. Nous montrons que ces performances ne dépendent pas seulement des applications, mais aussi des spécificités et paramètres des systèmes de STM. Pour étudier plus en détail ces questions et aider les développeurs à comprendre et améliorer les performances, nous proposons une approche générique pour la collecte des informations pertinentes sur les transactions. Notre solution peut être appliquée à des systèmes de STM et des applications différentes car elle ne modifie les codes sources ni des applications cibles, ni celui du système de STM. Nous montrons ensuite que l'information collectée peut être utile pour comprendre les performances des applications se synchronisant à l'aide de la STM. Ce travail a abouti à une publication dans les actes de la conférence *Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)* en 2011 :

- Márcio Castro, Kiril Georgiev, Vania Marangonzova-Martin, Jean-François Méhaut, Luiz Gustavo Fernandes et Miguel Santana. « Analysis and Tracing of Applications Based on Software Transactional Memory on Multicore Architectures ». Dans : *Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)*. Aya Napa, Cyprus : IEEE Computer Society, 2011, p. 199–206. isbn : 978-0-7695-4328-4. doi : 10.1109/PDP.2011.27

La deuxième contribution de cette thèse concerne la proposition d'une approche visant à améliorer les performances des applications de TM grâce à l'exploitation de la hiérarchie mémoire de plates-formes multicœurs modernes. Nous évaluons et démontrons que les stratégies de placement de *threads* (*thread mapping*) permetent de mieux utiliser les plates-formes multicœurs et donc d'améliorer les performances des applications de TM. Cependant, l'efficacité d'une telle approche repose sur l'appariement du comportement de l'application avec les caractéristiques du système de TM sous-jacent. En particulier, les systèmes de STM rendent cette tâche encore plus difficile en raison de leur système d'exécution. Les systèmes existants de STM mettent

en œuvre plusieurs mécanismes pour la détection et résolution des conflits. Ce qui conduit les applications TM à se comporter différemment pour chaque combinaison de ces mécanismes. Ainsi, la prédiction d'une stratégie de placement de *threads* appropriée pour une application/système de STM spécifique est une tâche délicate. Nous nous attaquons à ce problème en utilisant de l'Apprentissage Automatique (*Machine Learning* – ML) pour déduire automatiquement une stratégie de placement de *threads* pour les applications de TM. Notre approche prend en compte non seulement des caractéristiques de l'application de TM mais aussi le système de STM et la plate-forme multicœurs sous-jacents. Ce travail a abouti à une publication dans les actes de la conférence *High Performance Computing Conference (HiPC)* en 2011 :

- Márcio Castro, Luís Fabricio Wanderley Góes, Christiane Pousa Ribeiro, Murray Cole, Marcelo Cintra et Jean-François Méhaut. « A Machine Learning-Based Approach for Thread Mapping on Transactional Memory Applications ». Dans : *High Performance Computing Conference (HiPC)*. Bangalore, India : IEEE Computer Society, 2011, p. 1–10. ISBN : 978-1-4577-1949-3. DOI : 10.1109/HiPC.2011.6152736

Enfin, notre troisième contribution étend la précédente approche en proposant une stratégie appropriées de placement de *threads* de manière dynamique pour des applications de TM. Nous soutenons que des applications plus complexes feront usage de la TM dans l'avenir proche. Ces applications peuvent être composées de plusieurs phases d'exécution, chaque phase ayant un comportement potentiellement différent. Ainsi, au lieu de prévoir et d'appliquer une seule stratégie statique de placement de *threads*, nous utilisons des techniques de profilage pour récupérer des informations utiles lors de l'exécution des applications de TM. Cette technique nous permet de détecter des changements de comportement et d'appliquer une stratégie de placement de *threads* adéquate à chaque phase. Nous avons mis en œuvre cette approche dans un système de STM récent pour rendre cela transparent à l'utilisateur.

Les résultats préliminaires de notre troisième contribution ont été présentés et discutés au *Euro-TM Workshop on Transactional Memory (WTM)* (associé avec la conférence EuroSys 2012). Les soumissions ont été évaluées par les membres du projet

Transactional Memories : Foundations, Algorithms, Tools, and Applications (Euro-TM) [1].

► Márcio CASTRO, Luís Fabrício GÓES, Luiz Gustavo FERNANDES et Jean-François MÉHAUT. « Dynamic Thread Mapping Based on Machine Learning for Transactional Memory Applications ». Dans : *Euro-TM Workshop on Transactional Memory (WTM)*. Extended abstract. Avr. 2012

Une version étendue de ce travail a été accepté pour une présentation dans les actes de la conférence *International European Conference on Parallel and Distributed Computing (Euro-Par)* en 2012.

► Márcio CASTRO, Luís Fabrício GÓES, Luiz Gustavo FERNANDES et Jean-François MÉHAUT. « Dynamic Thread Mapping Based on Machine Learning for Transactional Memory Applications ». Dans : *International European Conference on Parallel and Distributed Computing (Euro-Par)*. T. 7484. Lecture Notes in Computer Science (LNCS). Rhodes Island, Greece : Springer-Verlag, 2012, p. 465–476. ISBN : 978-3-642-32819-0. DOI : 10.1007/978-3-642-32820-6_47

1. http://www.eurotm.org

CHAPITRE 2

Mémoire Transactionnelle : Une Nouvelle Approche de la Synchronisation

2.1 Les principes de base

La Mémoire Transactionnelle (*Transactional Memory* – TM) offre une nouvelle façon intéressante de développement d'applications parallèles en utilisant un niveau d'abstraction plus élevé. Elle déplace le problème de la synchronisation correcte au système de TM, qui est responsable pour s'assurer que : les interblocages ne se produisent pas, les situations de compétition (*race conditions*) sont correctement traitées et les blocagues sont effectués à une granularité qui permet de bien exploiter le parallélisme inhérent à l'application.

L'idée originale remonte à 1977, lorsque *D. Lomet* a remarqué qu'une abstraction similaire à celle de transactions de base de données pourrait être utilisé comme mécanisme de programmation afin d'assurer la cohérence des données partagées entre plusieurs *threads* [Lom77]. Seize ans plus tard, en 1993, *M. Herlih* et *J. Moss* ont proposé une implémentation d'un système de TM au niveau matériel [HM93]. Dès la publication de ce travail, nous observons un intérêt croissant de recherches sur la Mémoire Transactionnelle.

Dans cette section, nous présentons certains aspects importants de la Mémoire Transactionnelle. Tout d'abord, nous présentons les concepts généraux. Enfin, nous discutons des différentes approches pour sa mise en œuvre.

2.1.1 Concept

Les concepts de base de la TM sont illustrés dans la Figure 2.1. Les transactions sont des séquences d'instructions (délimitées par des blocs de code) exécutés par des *threads* [HS08]. La façon dont une transaction est définie dans le code source dépend de son implémentation dans le langage de programmation utilisé. Cependant, les transactions sont définies dans la plupart des compilateurs qui prennent en charge la TM par des blocs **atomiques**. Le code interne au bloc atomique (ainsi que les routines qu'il invoque) est exécuté comme une transaction et est donc assuré d'être exécuté de façon atomique et isolé indépendamment des situations de compétition (*data races*) [Dal+10 ; HLR10].

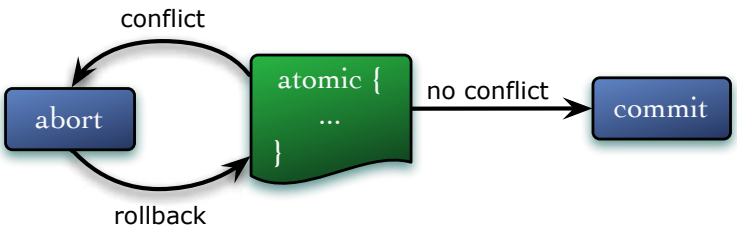


FIGURE 2.1 – Principes de base de la Mémoire Transactionnelle.

Lors de l'exécution d'une application, les opérations des blocs atomiques sont exécutées spéculativement et la Mémoire Transactionnelle controle les accès concurrents et détecte les conflits. Quand un conflit survient, une seule transaction impliquée dans le conflit sera validée (de l'anglais *commit*) alors que les autres seront annulées (de l'anglais *abort*) et ré-exécutées (de l'anglais *rollback*). Dans les cas des transactions sans conflit, le système de TM permettra que toutes les transactions soient validées simultanément.

À défaut de bonnes traductions en Français, nous emploierons dans la suite le terme *commit* pour désigner la validation d'une transaction et le fait que ses modifications soient visibles par tous les autres transactions, et le terme *abort* pour désigner

l'annulation d'une transaction suite à un conflit. Par extension, nous utiliserons les verbes **commiter** et **aborter**. Le terme transactionnel désigne ce qui a trait à une transaction. Ainsi, des données transactionnelles sont des données accédées durant une transaction.

2.1.2 Choix de conception

Fondamentalement, il existe quatre critères importants qui doivent être pris en compte lors de la conception des systèmes de TM : la granularité des transactions, la gestion des versions, la détection et la résolution des conflits.

La **granularité des transactions** est l'unité de stockage dans laquelle un système de TM détecte les conflits [HLR10]. Dans les langages orientées objet, il est courant d'utiliser la **granularité d'objet**, qui détecte les conflits lorsque les *threads* modifient l'état d'un même objet. Il existe également d'autres alternatives telles que la **granularité du mot mémoire** ou la **granularité d'une ligne de cache**. La première utilise le mot de mémoire comme l'unité de détection de conflit, alors que la dernière utilise un ensemble de mots chargés dans le cache. La granularité choisie peut évidemment affecter les performances des applications et influencer la mis en œuvre d'un système de TM.

Le second mécanisme important est la **gestion de version** qui est responsable pour contrôler la façon dont les modifications de données en mémoire sont gérées. En général, les systèmes de TM utilisent l'une de ces deux approches : la gestion de version précoce (de l'anglais *eager version management*) ou la gestion de version paresseuse (de l'anglais *lazy version management*). La première stocke les nouvelles valeurs en place et les anciennes valeurs dans un historique des valeurs. À l'inverse, la gestion de version paresseuse laisse les valeurs pré-transactionnelles en mémoire et place les nouvelles valeurs dans des copies privées.

Le troisième mécanisme important est la **détection des conflits**. Fondamentalement, la politique de détection des conflits dicte le moment où les conflits sont détectés. On distingue classiquement deux types de détection des conflits : la détection précoce des conflits (de l'anglais *eager conflict detection*) et la détection paresseuse des conflits (de l'anglais *lazy conflict detection*). Dans la première approche, les conflits sont détectés lorsqu'ils se produisent pendant l'exécution des transactions. En

revanche, la deuxième approche détecte les conflits lorsque la transaction commite. Cela permet de réduire les surcoûts liés à l'exécution des transactions au prix d'une augmentation du surcoût de la phase de commit.

Enfin, lorsqu'un conflit est détecté, un mécanisme de **résolution des conflits** est invoqué par le système de TM. Ce mécanisme décide quelle transaction doit être ré-exécutée. Cette tâche est généralement traitée par le **gestionnaire de contention** qui met en œuvre une ou plusieurs **politiques de résolution de conflits**. Les deux alternatives les plus utilisées sont d'aborter immédiatement l'une des transactions en conflit (dite *suicide*) ou de rajouter un temps d'attente après un abort, de manière à éviter la congestion engendrée par une transaction qui aborte et recommence continuellement (dite *backoff*). De la même façon, le choix de la politique de résolution des conflits peut également affecter les performances d'un système de TM.

2.1.3 Les systèmes de TM

Les choix de conception mentionnés ci-dessus constituent le cœur d'un système de TM. Cependant, les systèmes peuvent être implémentés en logiciel ou implantés en matériel.

Les systèmes de **Mémoire Transactionnelle Logicielle** (*Software Transactional Memory* – STM) mettent en œuvre toutes les sémantiques transactionnelles dans le logiciel, telles que la détection des conflits, la gestion des versions et la résolution des conflits. Elle ne requiert donc pas de support matériel particulier. Ces systèmes sont en général implémentés sous la forme d'une bibliothèque [HLR10].

À l'inverse des systèmes de STM, les systèmes de **Mémoire Transactionelle Materielle** (*Hardware Transactional Memory* – HTM) sont basés sur le fait qu'aucune instrumentation logicielle spécifique n'est requise, mais que le système est entièrement implanté au niveau matériel. Bien souvent, les systèmes de HTM modifient les protocoles de cohérence de *cache* pour mettre en œuvre les mécanismes de détection et de résolution de conflits [HM93 ; Moo+06].

Enfin, les systèmes de **Mémoire Transactionnelle Hybride** (*Hybrid Transactional Memory* – HyTM) utilisent une approche hybride dans laquelle le matériel sert simplement à optimiser les performances des transactions qui sont contrôlées par une solution logicielle [Kum+06 ; Shr+06].

2.2 Validation et évaluation de la TM

Des nombreux *benchmarks* ont été crées depuis la première proposition d'un système de TM. Les *benchmarks* les plus utilisés dans le domaine peuvent être classés dans trois groupes :

- ***Microbenchmarks* basés sur des structures de données**. Ils sont basés sur des structures de données simples pour évaluer les systèmes de TM. Ils sont utiles pour la construction de base au niveau des idées de modèles de TM et peuvent être configurés pour répondre aux besoins des utilisateurs. Des exemples de paramètres communs sont le pourcentage des insertions, des suppressions et des recherches à l'intérieur de la structure de données. Cependant, ils ne présentent pas un large éventail de caractéristiques de TM [Kes+09] et ne représentent pas des charges de travail réalistes. Comme exemples, on peut citer l'arbre bicolore [DSS06], la table de hachage [Dic+09], la liste chaînée [FFR08] et la liste à enjambements [Dra+11].
- ***Benchmarks* réalistes.** Ils représentent de vraies charges de travail originaires d'une grande variété d'algorithmes et domaines d'application différents. Ces applications ont un large éventail de comportements transactionnels : différentes tailles de transactions, différents taux de conflits, des transactions à gros grains et à grains fins, etc. Le *benchmark* les plus utilisé est le *Stanford Transactional Applications for Multi-Processing* (STAMP) [Min+08] qui comprend 8 applications et 30 variantes de paramètres d'entrée.
- **Générateurs des charges de travail.** Les générateurs sont utiles pour explorer un large éventail de comportements transactionnels et stresser les aspects particuliers d'un système de TM. Ils ont généralement plusieurs paramètres de configuration qui peuvent être facilement modifiés pour produire des charges de travail. Ils ont aussi la capacité d'imiter le comportement des applications réelles de TM. Le générateur le plus utilisé est EigenBench [Hon+10]. Il permet aux utilisateurs d'effectuer une exploitation approfondie de l'espace orthogonal des caractéristiques des applications de TM. En raison de ses nombreux paramètres, il peut être facilement configuré pour imiter une grande variété de charges de travail (comme par exemple les charges de travail présentes dans STAMP).

CHAPITRE 3

Analyse des Performances des Applications de Mémoire Transactionnelle

3.1 Impact de la STM sur les performances des applications

Nous avons mentionné précédemment que la performance d'une application de TM peut être affectée par le système de STM sous-jacent. Du côté du système de STM, cet impact provient de ses choix de conception, telles que la gestion des versions et la détection des conflits [Wan+11] et des mécanismes de résolution [HYH12]. Comme les systèmes de STM utilisent des différentes stratégies, nous nous attendons à ce que les applications de TM se comportent différemment selon le système de STM sous-jacent.

Afin d'étudier cet impact, nous avons effectué des expériences avec toutes les applications du *benchmark* STAMP. Ce *benchmark* offre des avantages importants par rapport à d'autres *benchmarks* : les applications utilisent une variété d'algorithmes qui appartiennent à différents domaines. En autre, les applications peuvent être facilement exécutées avec des systèmes de STM différents. Afin de stresser les systèmes de STM, nous avons choisi les plus grandes tailles de paramètres d'entrée

décrites dans l'article de STAMP [Min+08]. Pour couvrir un plus large éventail de caractéristiques transactionnelles, nous avons utilisé les paramètres de contention bas pour les applications vacation et kmeans.

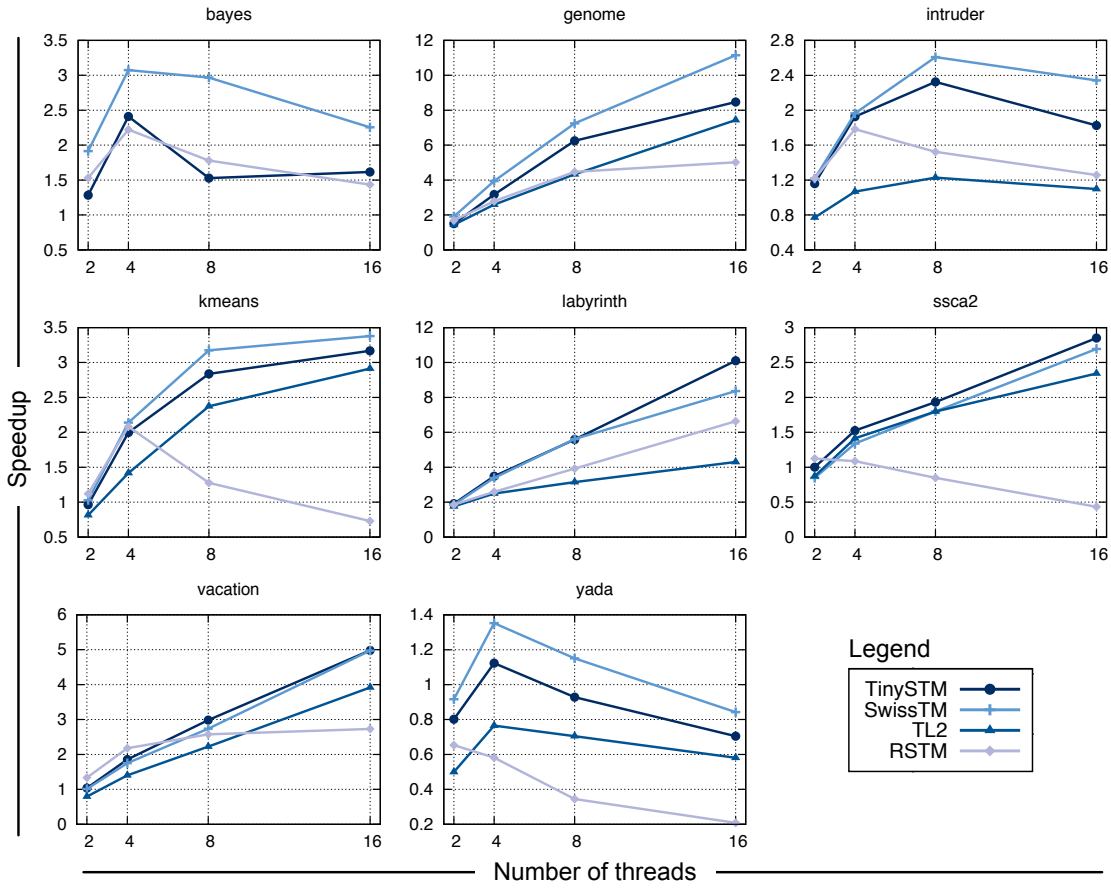


FIGURE 3.1 – Speedups des applications de STAMP avec quatre systèmes de STM.

Nous avons exécuté les applications de STAMP avec quatre systèmes de STM récents. Les systèmes utilisés sont TinySTM [FFR08] (version 1.0.3), TL2 [DSS06] (version 0.9.6), SwissTM [DGK09] (version 15.08.2011) et RSTM [Mar+06] (version 7). Ils représentent les systèmes de STM les plus connus implémentés en langage C (TinySTM et TL2) et C++ (SwissTM et RSTM). Comme certains systèmes de STM peuvent être configurés pour utiliser des différents algorithmes, nous avons utilisé leurs configurations par défaut. Chaque expérience a été exécutée au moins 30 fois et les *speedups* ont été calculés en utilisant le temps moyen d'exécution sur la plate-

forme SMP-24 (nous présentons la description détaillée de cette plate-forme dans la Section 5.1). Il est également important de mentionner que toutes les expériences ont été réalisées avec un accès exclusif à la plate-forme et nous avons utilisé une stratégie de placement de *threads* statique pour éviter les migrations des *threads*. Cette stratégie fixe un *thread* par cœur et place les *threads* aussi proches que possible afin de profiter du partage de *cache*.

La Figure 3.1 présente les *speedups* de toutes les applications avec les quatre systèmes de STM [1]. Comme l'on peut constater, les performances des applications peuvent être considérablement affectées en fonction du système de STM utilisé. Dans l'ensemble, TinySTM et SwissTM ont présenté les meilleurs résultats. L'application bayes a présenté un comportement imprévisible. Principalement avec TinySTM, le temps d'exécution a varié considérablement ce qui rend difficile la comparaison des ces performances avec SwissTM. Vacation et ssca2 ont présenté des performances très similaires.

Les deux cas les plus intéressants étaient intruder et labyrinth. Dans le cas de l'application intruder, les performances obtenues avec ces deux systèmes de STM ont été très similaires avec un petit nombre de *threads*. Toutefois, avec 8 et 16 *threads*, le système SwissTM a obtenu des meilleurs résultats. Dans le cas de l'application labyrinth, nous avons remarqué un comportement similaire mais cette fois-ci le système TinySTM a été plus performant que SwissTM avec un grand nombre de *threads*.

Nous pensons que l'utilisation de techniques de traçage peut s'avérer utile pour analyser ces impacts. Dans la Section 3.2, nous présentons une approche générique pour récupérer des informations pertinentes à la TM. Enfin, dans la Section 3.3, nous utilisons cette approche pour mieux comprendre la performance de ces deux cas intéressants.

3.2 Vers un mécanisme de traçage adaptée à la TM

Dans cette thèse, nous nous sommes particulièrement intéressés par les événements provenants de l'utilisation de la STM. Cela nous permet de proposer une

1. Nous n'avons pas inclus les résultats de bayes avec TL2, puisque cette configuration conduit à des exécutions incorrectes de cette application.

approche capable de récupérer des informations utiles à faible intrusivité et indépendante de l'application et du système de STM.

Les systèmes de STM offrent un ensemble d'opérations qui fournissent des fonctionnalités de base de la TM. Ces opérations sont implémentées comme des fonctions dans les systèmes de STM et elles apparaissent dans le code source de l'application en tant qu'appels aux fonctions du système de STM. Dans la Table 3.1, nous définissons les événements les plus importants qui peuvent être générés à partir d'un système de STM. Nous avons également corrélé ces événements avec leurs fonctions correspondantes dans TinySTM [2].

Événement	**Fonction (TinySTM)**	**Action**
StmInit	`stm_init()`	Initialiser le système de STM.
StmExit	`stm_exit()`	Finaliser le système de STM.
TxBegin	`stm_start()`	Début d'une transaction.
TxEnd	`stm_commit()`	Commit d'une transaction.
TxAbort	`stm_rollback()`	Ré-exécution d'une transaction abortée.
TxRead	`stm_load()`	La transaction lit une donnée en mémoire.
TxWrite	`stm_store()`	La transaction écrit une donnée en mémoire.

TABLE 3.1 – Fonctionnalités de base d'un système de STM.

Le traçage des événements cités ci-dessus nous permet de recueillir des informations utiles de l'exécution des applications à base de TM. Par exemple, les données tracées peuvent être utilisées pour détecter si l'application présente des points de contention ou si cette contention est répartie sur toute l'exécution.

Nous avons implémenté notre solution de traçage (`libTraceSTM`) en s'appuyant sur le mécanisme d'édition de liens dynamique de Linux. Ce mécanisme offre un moyen simple d'intercepter les appels aux fonction d'une application sans modifier ces codes sources originaux. Lors de l'exécution, notre mécanisme de traçage intercepte les fonctions souhaitées et redirige l'exécution vers les fonctions définies dans le traceur (dites *wrappers*). Les fonctions *wrapper* sont implementées par l'utilisateur.

La Figure 3.2 présente le fonctionnement du traceur. Quand chaque *thread* est initialisé par l'application de TM, le traceur crée automatiquement une structure

2. Il n'existe pas de norme en matière de noms de fonctions (API). Bien que le nom de ces fonctions peut varier en fonction du système de STM, tous les systèmes ont généralement telles fonctionnalités.

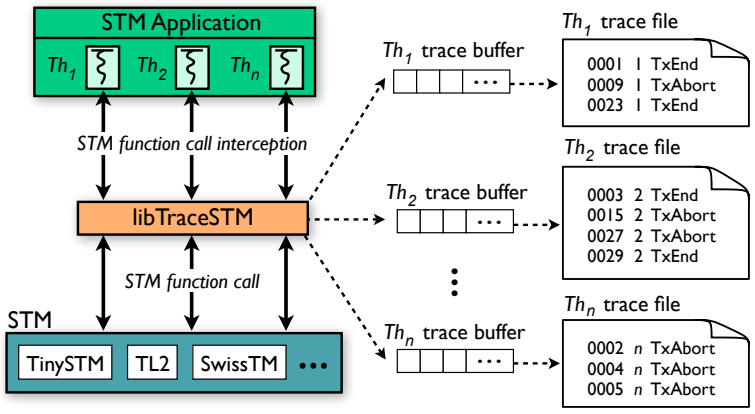


FIGURE 3.2 – Mécanisme de traçage.

de données interne pour stocker les informations sur ce *thread* (un fichier de trace et une mémoire tampon pour stocker temporairement des événements tracés). Les traces des événements contiennent non seulement l'identifiant du *thread* mais aussi leurs estampillages, ce qui permet d'analyser temporellement les événements générés par des applications basées sur la TM.

Les événements tracés sont tout d'abord stockées dans la mémoire tampon. Lorsque la mémoire tampon est pleine, son contenu est vidé dans le fichier de trace correspondant. Puisque nous obtenons un fichier de trace par *thread*, les fichiers de trace individuels doivent être fusionnés en un seul fichier suivant un ordre global d'événements. Nous fournissons donc un outil qui effectue un tri fusion des fichiers de trace individuels.

3.3 Études de cas

Dans cette section, nous appliquons notre mécanisme de traçage sur des applications de TM du *benchmark* STAMP. L'objectif est d'utiliser des techniques de visualisation simples pour analyser et comprendre le comportement des applications à partir des données tracées.

Puisque notre mécanisme de traçage enregistre les estampilles de chaque événement, nous pouvons savoir quand chaque événement s'est produit lors de l'exécution de l'application. Ceci est utile pour trouver des points de contention pendant

l'exécution ou pour vérifier si cette contention est répartie sur toute l'exécution. Cette information peut être obtenue à partir du traçage des événements $TxEnd$ et $TxAbort$. Nous pensons que cet ensemble réduit d'événements nous permettra de recueillir des informations pertinentes sans augmenter le degré d'intrusion.

Nous avons sélectionné les applications intruder et labyrinth pour nos études de cas car elles présentent différents niveaux de contention, des différentes performances et des comportements distincts fonction du système de STM utilisé. Les systèmes de STM choisis pour cet étude sont TinySTM et SwissTM.

3.3.1 Intruder

Dans notre première étude de cas, nous voulons comprendre les causes de la mauvaise performance de l'application intruder avec des nombres de *threads* élevés. En plus de cela, nous voulons savoir pourquoi SwissTM a eu des résultats relativement meilleurs que TinySTM.

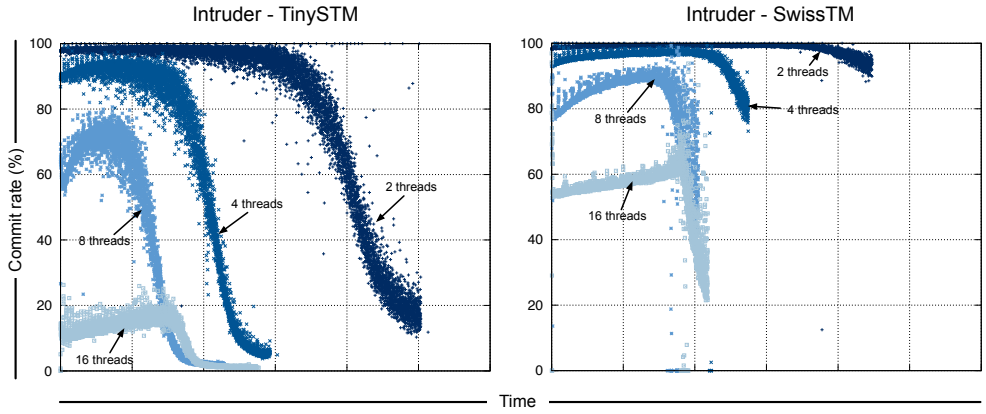


FIGURE 3.3 – Taux de commits de l'application intruder avec TinySTM et SwissTM.

La Figure 3.3 compare le taux de commit de cette application avec TinySTM et SwissTM. Intruder exécute un très grand nombre de transactions (environ 255 millions de transactions avec 16 *threads*), ce qui génère 4 Go de données dans notre format binaire. Tout d'abord, les résultats montrent que les taux de commit changent considérablement durant l'exécution et ils sont fortement liés au nombre de *threads*. Nous avons également observé que cette application présente une augmentation de la contention vers la fin de son exécution. Le système SwissTM a eu des taux de

commits toujours plus élevés par rapport à ceux de TinySTM. Cela veut dire que le gestionnaire de contention de SwissTM est plus adapté à cette application, ce qui résulte à des meilleures performances.

Nous croyons que la raison pour laquelle SwissTM a été plus performant que TinySTM est dû au fait que TinySTM utilise la politique *suicide* pour la résolution de conflits. Cette politique se comporte bien sur des charges de travail à faible contention, ce qui n'est pas le cas de l'application intruder.

3.3.2 Labyrinth

Dans notre second étude de cas, nous analysons l'application labyrinth avec TinySTM et SwissTM. Comme nous l'avons pu observer dans la Section 3.1, les performances de labyrinth avec TinySTM et SwissTM ont été similaires avec un faible nombre de *threads*. Toutefois, avec 16 *threads* le système TinySTM a été plus performant que SwissTM.

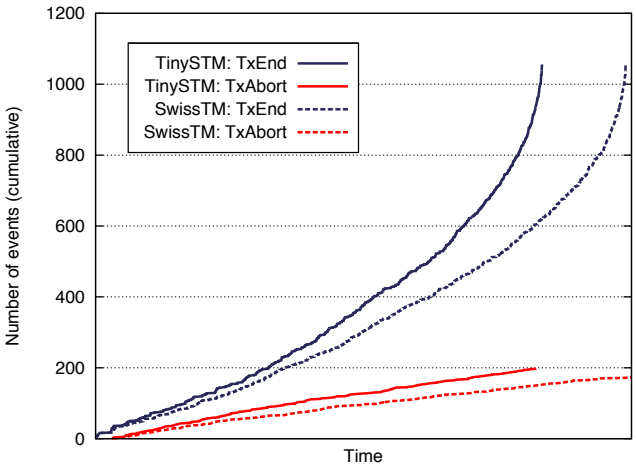


FIGURE 3.4 – Nombre de $TxEnd/TxAbort$ accumulés dans labyrinth avec 16 *threads*.

Dans la Figure 3.4, chaque courbe représente le nombre accumulé d'événements ($TxEnd$ ou $TxAbort$) au cours de l'exécution de labyrinth avec 16 *threads* sur les systèmes TinySTM et SwissTM. Les résultats montrent que le taux de croissance des événements $TxEnd$ est exponentielle et est très similaire sur les deux systèmes de STM. Par ailleurs, le taux de croissance de $TxAbort$ est linéaire mais aussi très

similaire sur les deux systèmes de STM. Cela indique que cette application prend avantage de l'approche optimiste de la TM et justifie les bonnes performances présentés dans la Section 3.1. Toutefois, le taux de croissance de transactions commitées est légèrement plus élevé sur TinySTM. Cela est dû au fait que TinySTM utilise la politique *suicide* pour la résolution de confits. Cette politique est plus performante dans le cas de l'application labyrinth car elle présente très peu de conflit et les transactions en conflit sont abortées immédiatement après la détection de conflit.

CHAPITRE 4

Optimisation des Performances des Applications de Mémoire Transactionnelle

4.1 Impact du placement de threads sur les performances

Les performances des applications parallèles peuvent être améliorées grâce à une bonne exploitation de la hiérarchie mémoire de plates-formes multicœurs modernes. Le placement de *threads* (*thread mapping*) est une des techniques qui permettent de mieux utiliser les ressources comme les *bus* mémoire, la mémoire principale et la mémoire cache.

Définir une stratégie efficace de placement de *threads* pour des applications de TM est un défi. Cela provient du fait que chaque système de STM met en œuvre des différents mécanismes pour détecter et résoudre les conflits entre les transactions. Ces mécanismes modifient le comportement de l'application de TM et influencent la décision de la stratégie de placement de *threads*.

Pour supporter cette affirmation, nous avons effectué plusieurs expériences avec toutes les applications du *benchmark* STAMP sur quatre systèmes de STM différents (TinySTM, SwissTM, TL2 et RSTM) et quatre stratégies de de placement de *threads*

(compact scatter, round-robin et linux). Fondamentalement, ces stratégies peuvent être caractérisées de la façon suivante :

- **scatter** : les *threads* sont répartis sur des différents processeurs. Cela permet d'éviter le partage de la mémoire *cache* entre les cœurs afin de réduire la contention sur la même *cache*.
- **compact** : les *threads* sont physiquement placés proches les uns des autres. Cela permet de réduire la latence d'accès à la mémoire car les *threads* partagent tous les niveaux de la hiérarchie de cache.
- **round-robin** : cette stratégie est une solution intermédiaire dans laquelle les *threads* partagent des niveaux plus hauts de cache (*e.g.*, L3), mais pas les plus bas (*e.g.*, L2).
- **linux** : c'est la stratégie utilisée par défaut sur Linux.

La Table 4.1 présente la différence en terme des temps d'exécution entre la meilleur et la pire stratégie de placement de *threads* pour toutes les applications du *benchmark* STAMP. Nous avons constaté que l'impact de ces stratégies peut être élevé selon l'application et le système de STM.

Application	TinySTM				SwissTM			
	Meilleur		Pire		Meilleur		Pire	
	stratégie	*temps (s)*	*stratégie*	*temps (s)*	*stratégie*	*temps (s)*	*stratégie*	*temps (s)*
bayes	compact	6.8	scatter	8.9	scatter	6.1	compact	9.6
genome	compact	4.0	scatter	10.2	scatter	1.5	compact	1.9
intruder	compact	44.9	linux	65.7	compact	15.7	scatter	16.9
kmeans	compact	6.7	scatter	7.8	round-robin	5.7	compact	6.6
labyrinth	scatter	15.6	round-robin	20.3	scatter	14.5	round-robin	15.7
ssca2	scatter	7.3	compact	9.6	scatter	7.5	compact	10.0
vacation	round-robin	7.7	linux	10.7	scatter	8.0	compact	9.8
yada	compact	13.1	scatter	17.1	compact	11.2	scatter	14.1

Application	TL2				RSTM			
	Meilleur		Pire		Meilleur		Pire	
	stratégie	*temps (s)*	*stratégie*	*temps (s)*	*stratégie*	*temps (s)*	*stratégie*	*temps (s)*
bayes	–	–	–	–	compact	7.8	round-robin	8.8
genome	scatter	2.2	compact	2.6	compact	2.5	linux	2.8
intruder	compact	34.5	scatter	39.6	compact	25.5	scatter	34.9
kmeans	round-robin	7.1	scatter	8.4	compact	11.8	scatter	17.2
labyrinth	round-robin	22.9	scatter	25.7	scatter	17.6	compact	19.6
ssca2	compact	11.8	scatter	15.3	compact	20.7	scatter	35.6
vacation	scatter	9.5	compact	10.4	compact	9.7	linux	13.3
yada	compact	16.3	scatter	20.0	compact	36.9	scatter	45.8

TABLE 4.1 – Impact du placement de *threads* sur les performances des applications.

4.2 Une approche basée sur l'apprentissage automatique pour le placement de threads

Pour faire face à la grande diversité des applications, des systèmes de STM et des plates-formes, nous proposons une approche basée sur l'Apprentissage Automatique (*Machine Learning* – ML) pour prédire automatiquement les stratégies de placement de *threads* appropriées pour les applications de TM.

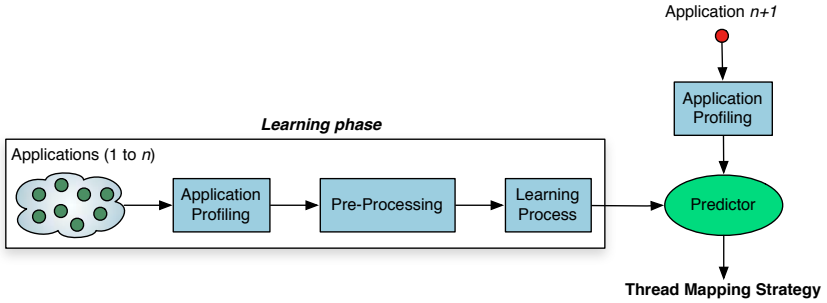


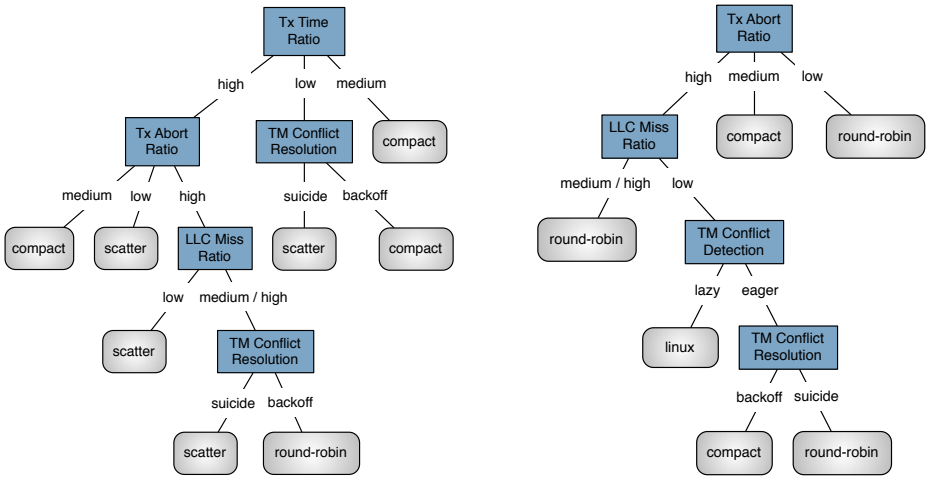
FIGURE 4.1 – Approche basée sur l'apprentissage automatique.

Notre approche est illustrée dans la Figure 4.1. Au cours d'une phase d'apprentissage préliminaire, nous construisons les profils des applications s'exécutant sur différents systèmes de STM. Pour construire les profils nous utilisons l'ensemble de métriques présentés dans la Table 4.2. Les données collectées sont utilisées par un algorithme de ML pour créer notre prédicteur. L'algorithme utilisé est le *Dichotomiser 3* (ID3) [Qui86]. Cet algorithme génère une arbre de décision qui sera utilisé comme un prédicteur de placement de *threads*.

Métrique	**Valeurs**	**Description**
Temps transactionnel	(0.0 ; 1.0)	Temps transactionnel / Temps total
Taux d'abort	(0.0 ; 1.0)	Nombre d'aborts / Nombre de commits + Nombre d'aborts
Détection de conflits	eager, lazy	STM conflict detection policies
Résolution de conflits	suicide, backoff	STM conflict resolution policies
Défauts de cache (LLC)	(0.0 ; 1.0)	Nombre de défauts de cache misses / Nombre d'accès
Placement de *threads*	compact, scatter, round-robin, linux	Stratégies de placement de *threads*

TABLE 4.2 – Métriques utilisées pour construire les profils des applications.

Une fois que la phase d'apprentissage est terminée, le prédicteur peut être utilisé pour déduire et ensuite appliquer une stratégie efficace de placement de *threads* à des nouvelles applications. Il est important de noter que la phase d'apprentissage n'est plus nécessaire. Au lieu de cela, le profilage est nécessaire afin de récuperer des informations de la nouvelle application. Les données collectées sont converties en conformité avec le format d'entrée de l'arbre de décision. Enfin, l'arbre de décision est alimenté par ces informations. L'arbre est ensuite exploré pour déterminer une stratégie de placement. La stratégie de placement de *threads* est donc appliquée.



(a) SMP-24 (b) SMP-16

FIGURE 4.2 – Arbres de décision générées par l'algorithme ID3 sur deux différentes plates-formes.

Nous avons utilisé toutes les applications du *benchmark* STAMP pour construire le prédicteur. Nous pensons que ces applications réalistes peuvent être utiles pour construire un prédicteur capable de capturer le comportement des applications de TM. La Figure 4.2 montre les deux arbres de décision générés par l'algorithme ID3 sur deux plates-formes : SMP-24 (à gauche) et SMP-16 (à droite). Les détails de ces plates-formes sont présentes dans la Section 5.1. Sur la plate-forme SMP-24, compact et scatter sont les deux plus importantes stratégies alors que round-robin et compact sont les plus importantes sur la plate-forme SMP-16. Nous pouvons également observer que l'arbre de décision de la SMP-16 est beaucoup plus simple

que celle de la SMP-24. Ceci peut être expliqué par le fait que la SMP-24 a une hiérarchie de *cache* plus complexe.

4.3 Placement statique de threads

Une fois que le prédicteur a été crée, il peut désormais prédire des stratégies de placement de *threads* pour des nouvelles applications de TM. Nous pouvons utiliser deux stratégies différentes pour appliquer le placement de *threads* d'une façon statique :

- **Prédiction après une exécution préliminaire** : la nouvelle application est exécutée et profilée durant toute son exécution. Puis, l'arbre de décision est décoré avec les informations collectées. L'arbre est ensuite exploré pour déterminer la stratégie de placement. La stratégie choisie par le prédicteur sera donc appliquée sur les exécutions ultérieures de cette application.
- **Prédiction lors de l'exécution après une courte période de profilage** : la nouvelle application démarre avec une stratégie de placement de *threads* par défaut et est profilée pendant une courte période de temps. Puis, l'arbre de décision est décoré avec les informations collectées. L'arbre est ensuite exploré pour déterminer la stratégie de placement. La stratégie choisie par le prédicteur sera donc appliquée à l'exécution.

4.4 Placement dynamique de threads

L'approche statique qui utilise le prédicteur lors de l'exécution après une courte période de profilage peut être adaptée pour fonctionner d'une façon dynamique. L'idée est d'effectuer le profilage ainsi que la prédiction à plusieurs reprises durant l'exécution de l'application. Comme la plupart des caractéristiques considérées peuvent varier au cours de l'exécution des applications composées de plusieurs phases, il est important de définir comment et quand le profilage va se produire. Nous proposons d'utiliser l'échantillonnage au lieu de profiler toute l'exécution de l'application. Cela réduit considérablement le coût du profilage. Notre proposition est illustrée dans la Figure 4.3.

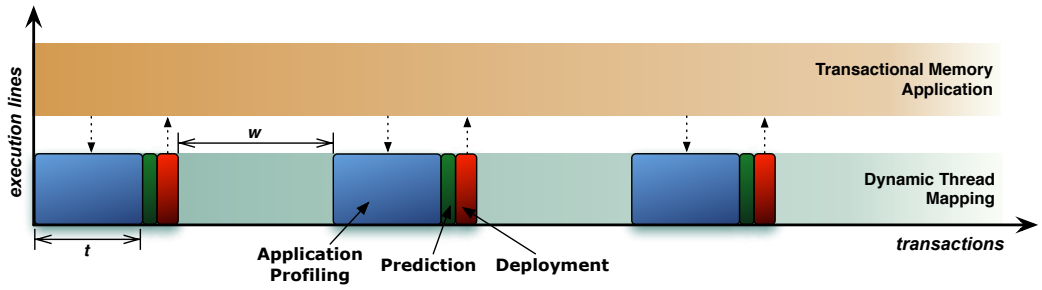


FIGURE 4.3 – Exécution avec le placement dynamique de *threads*.

Notre approche fonctionne de la façon suivante. Au cours de l'initialisation de l'application, nous appliquons la stratégie de placement de *threads* par défaut de Linux. Ensuite, nous profilons t transactions committées lors de l'exécution pour récupérer les informations nécessaires à notre prédicteur. L'information profilée est utilisée par le prédicteur pour sélectionner une stratégie de placement de *threads* adaptée aux caractéristiques de la charge de travail actuelle. La stratégie est donc déployée et reste inchangée pendant les w prochaines transactions committées. Ce processus est répété jusqu'à ce que l'application se termine.

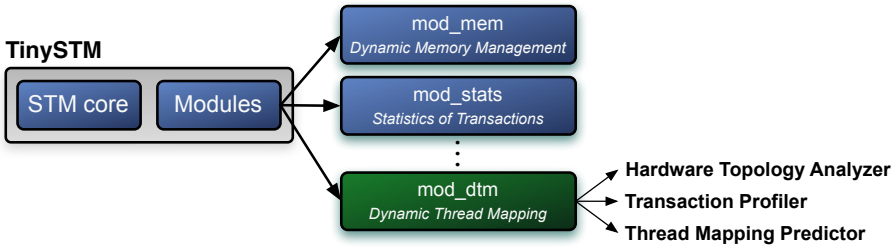


FIGURE 4.4 – Implementation du placement dynamique dans TinySTM.

Pour que notre solution soit transparente pour les utilisateurs, nous l'avons implementée dans un système de STM récent. Nous avons choisi TinySTM parmi les systèmes de STM car il est léger, efficace et a une structure modulaire qui peut être facilement étendue avec de nouvelles fonctionnalités (Figure 4.4).

Fondamentalement, le système TinySTM est composé d'un noyau, dans lequel la plupart du code du système de STM est implementé ainsi que quelques modules supplémentaires. Ces modules implémentent des fonctionnalités de base telles que la gestion de la mémoire dynamique (`mod_mem`) et les statistiques des transactions

(`mod_stats`). Nous avons ajouté un nouveau module appelé `mod_dtm` qui permet TinySTM de réaliser le placement dynamique de *threads* d'une façon transparente. Notre module contient les composants suivants :

- L'analyseur de topologie (*Hardware topology analyzer*) : est chargé de récupérer des informations utiles de la plate-forme sous-jacente (*i.e.*, la hiérarchie de *caches* et comment elles sont partagées). Pour cela, nous utilisons la bibliothèque *Hardware Locality* (hwloc) library [Bro+10].
- Le prédicteur (*Thread mapping predictor*) : s'appuie sur l'arbre de décision pour prédire la stratégie de placement de *threads*. Pendant la phase de prédiction, l'arbre est parcourue et la stratégie de placement de *threads* est ensuite déployée.
- Le mécanisme de profilage de transactions (*Transaction profiler*) : effectue le profilage de l'application. Les défauts de cache sont obtenus par la bibliothèque *Performance Application Programming Interface* (PAPI) [Ter+10]. Les autres métriques sont obtenues par l'utilisation de compteurs et des informations provenants du système de STM.

CHAPITRE 5

Expérimentations, Évaluation et Analyse

5.1 Plates-formes multicœurs

Nous avons mené nos expériences sur deux plates-formes multicœurs (Table 5.1). La **SMP-24** est une plate-forme multicœur basée sur quatre processeurs Intel Xeon X7460 à six cœurs. Chaque paire de cœurs partage une mémoire *cache* L2 (3 Mo) et chaque processeur a une cache L3 partagée (16 Mo).

	Characteristic	**SMP-24**	**SMP-16**
Matériel	Processeur	Intel Xeon X7460	Intel Xeon E7320
	Nombre de cœur	24	16
	Nombre de processeurs	4	4
	Fréquence (GHz)	2.66	2.13
	Cache (Mo)	16 (L3)	2 (L2)
	Mémoire RAM (Go)	64	64
Logiciel	**Nom / version**	**SMP-24**	**SMP-16**
	Noyau Linux	3.2.0-2	2.6.18
	GCC	4.6.3	4.1.2
	TinySTM	1.0.3	1.0.3
	Benchmark STAMP	0.9.10	0.9.10
	Eigenbench	0.8.0	–

TABLE 5.1 – Caractéristiques des plates-formes.

La **SMP-16** est une plate-forme multicœur basée sur quatre processeurs Intel Xeon

E7320 à quatre cœurs. Chaque paire de cœurs partage une mémoire cache L2 (2 Mo). Toutes les expériences ont été réalisées avec un accès exclusif à ces plates-formes.

Nous avons utilisé le système TinySTM pour toutes nos expériences car c'est un système léger, performant et peut être facilement configuré pour utiliser des différentes politiques pour la détection et résolution de conflits.

5.2 Placement statique de threads

Pour l'evaluation du placement statique de *threads*, nous avons utilisé la prédiction lors de l'exécution après une courte période de profilage. La nouvelle application démarre avec une stratégie de placement de *threads* par défaut et est profilée pendant une courte période de temps. Puis, l'arbre de décision est décoré avec les informations collectées. L'arbre est ensuite exploré pour déterminer la stratégie de placement. La stratégie choisie par le prédicteur est donc appliquée à l'exécution.

Nous avons utilisé les applications de STAMP pour le processus d'apprentissage. Nous avons également utilisé deux politiques de détection de conflits (eager et lazy) et de résolution de conflits (suicide et backoff) visant à alimenter le processus d'apprentissage. Nous évaluons le prédicteur en utilisant la méthode *leave-one-out-cross-validation*. Dans cette méthode, nous excluons l'application que nous voulons tester du processus d'apprentissage. Cela signifie que l'arbre de décision utilisé par le prédicteur ne contient pas d'information sur l'application à être testée. La prédiction est faite lors de l'exécution après une courte période de profilage.

La Figure 5.1 compare les performances de chaque stratégie placement de *threads* et notre approche basée sur l'apprentissage automatique avec la performance de l'oracle. L'oracle représente la performance maximale qui peut être obtenue, car il considère la meilleure stratégie pour chaque combinaison de paramètres. Une combinaison de paramètres est une instance d'une application/configuration du système de STM/plate-forme. Dans la dernière colonne, nous présentons également les performances globales de chaque stratégie.

Dans l'ensemble, l'approche basée sur le ML a fait des bonnes prédictions sur les deux plates-formes. Sur ces plates-formes, l'approche basée sur le ML a atteint 97,44% (SMP-24) et 97,04% (SMP-16) de la performance de l'oracle. Sur la plate-forme SMP-24, linux était la pire stratégie et a atteint 89,27% de la performance de

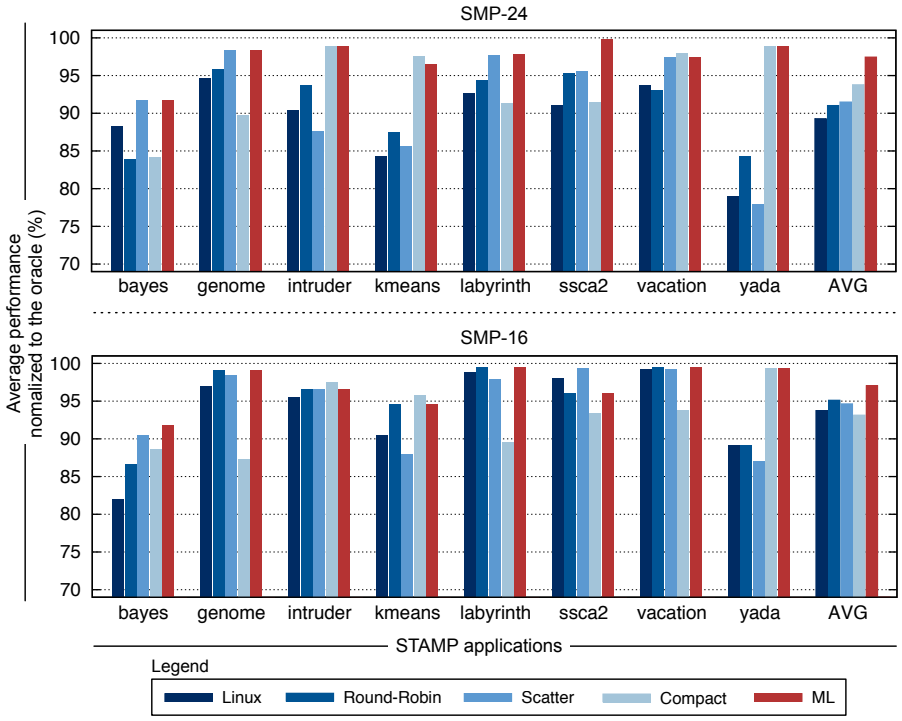


FIGURE 5.1 – Performances l'approche basée sur l'apprentissage automatique.

l'oracle. Différemment, compact était le pire sur la plate-forme SMP-24, réalisant 93,14% de la performance de l'oracle.

5.3 Placement dynamique de threads

Comme la plupart des transactions au sein de chaque application du *benchmark* STAMP ont généralement un comportement très similaire, ces applications ne sont pas appropriées pour l'évaluation de notre approche dynamique. Pour cette raison, nous avons utilisé EigenBench [Hon+10] pour créer de nouvelles applications de TM composées de différentes charges de travail. Ces charges de travail ont différentes caractéristiques comme la taille de transactions (grandes et petites), la probabilité de conflits (haute et basse) et le temps passé dans des transactions (long et court). Comme nous supposons deux possibles valeurs discrètes pour chacun de ces paramètres, nous pouvons créer un total de 2^3 charges de travail (nommées

$W_1, W_2, \ldots, W_8$).

Ensuite, nous avons établi un ensemble d'applications à partir des 8 charges de travail distinctes. Nous avons fixé le nombre de phases à 3, donc chaque application sera composé de trois charges de travail. Par conséquent, toutes les applications possibles composées de trois charges de travail distinctes est déterminée par $C_n^k = C_8^3$, *i.e.*, 56 applications ($A_1, A_2, \ldots, A_{56}$). Ainsi, l'ensemble des applications peuvent être représentées comme suit : $A_1 = \{W_1, W_2, W_3\}$, $A_2 = \{W_1, W_2, W_4\}$, ..., $A_{56} = \{W_5, W_6, W_7\}$. Les phases (charges de travail) ont été parallélisées à l'aide de la bibliothèque Pthreads.

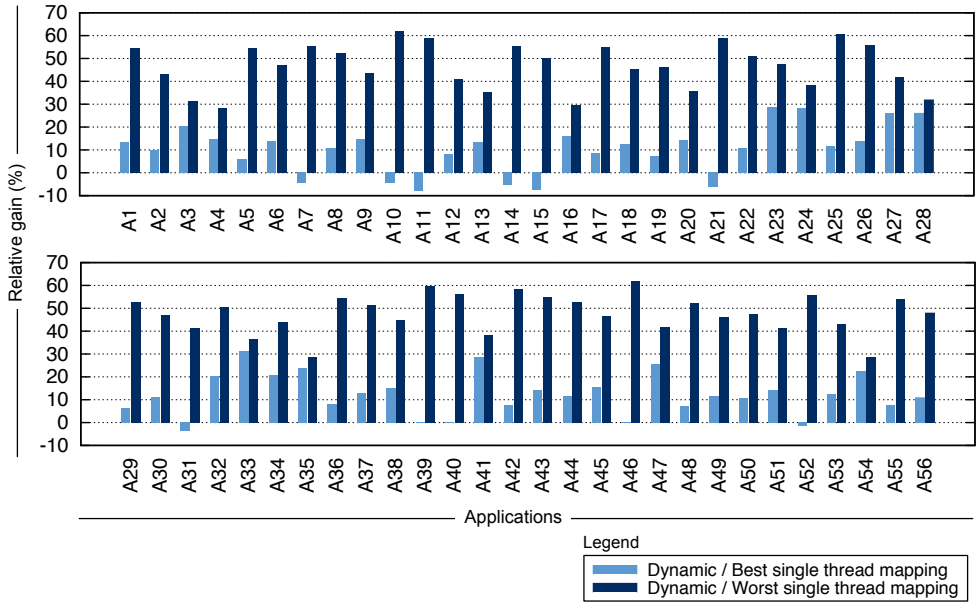


FIGURE 5.2 – Gains de l'approche dynamique par rapport à la meilleur/pire stratégie statique.

Nous avons exécuté toutes les applications avec les stratégies de placement de *threads* statiques (compact, scatter and round-robin), linux et notre approche dynamique. La Figure 5.2 présente les gains relatifs de l'approche dynamique par rapport à la meilleur/pire stratégie statique. Le gain relatif est donné par $1 - \frac{\overline{x}_d}{\overline{x}_s}$, où $\overline{x}_d$ et $\overline{x}_s$ sont les temps moyens obtenus à partir d'au moins 30 exécutions en utilisant la stratégie dynamique et la meilleur/pire stratégie statique respectivement. Ainsi, les valeurs positives signifient des gains de performance alors que les valeurs négatives

signifient des pertes. L'approche dynamique a présenté des meilleurs résultats que les stratégies statiques. Cela est dû au fait que l'approche dynamique permet de choisir une stratégie adaptée à chaque phase.

CHAPITRE 6

Conclusions et Perspectives

Les fabricants de processeurs ont répondu à la demande croissante de puissance de calcul en fournissant des processeurs de plus en plus rapides. Cependant, obtenir les meilleures performances sans augmenter ni la consommation d'énergie ni la production de chaleur est récemment devenue une préoccupation majeure. Pour répondre à cette nouvelle exigence, les fabricants investissent maintenant sur les plates-formes multicœurs, en s'appuyant sur le fait que de bonnes performances de calcul et de consommation d'énergie peuvent être obtenues si l'on réduit les vitesses d'horloge des unités de calcul (cœurs), tout en augmentant le nombre de cœurs.

Dans ce contexte, la Mémoire Transactionnelle Logicielle est une nouvelle approche rendant plus facile le développement d'applications parallèles sur les plates-formes multicœurs modernes. Ce modèle de programmation permet au programmeur d'écrire des portions parallèles de code dans des transactions. Ces transactions sont exécutées de façon atomique et isolées les unes des autres, tout en conservant une correcte exécution parallèle. Lors de l'exécution, les transactions sont exécutées spéculativement et le système d'exécution de la TM trace en permanence les accès concurrents et détecte les conflits. Les conflits éventuels sont alors résolus par la ré-exécution des transactions impliquées dans le conflit. Grâce à cela, la synchronisation correcte des accès concurrents aux données partagées n'est plus à la charge du programmeur. Cependant, tout comme les autres environnements de haut niveau précédemment cités, la STM cache des aspects importants qui peuvent avoir une inci-

dence sur les performances des applications utilisant de la Mémoire Transactionnelle. Par ailleurs, les applications utilisant de la TM peuvent se comporter différemment en fonction des caractéristiques du système de STM sous-jacent.

Les contributions de cette thèse portent sur l'analyse et l'optimisation des performances des applications reposant sur de la TM pour des plates-formes multicœurs en utilisant des techniques à faible intrusivité.

6.1 Contributions

Dans un premier temps, nous avons montré que l'analyse de performances des applications de TM est rendue complexe par le fait que le modèle de TM et le système de STM masquent de nombreux aspects pertinents. De plus, la diversité des applications basées sur de la TM et des systèmes de STM rend difficile d'établir une solution simple et générique qui permet aux développeurs d'accomplir cette tâche. Pour s'attaquer à ce problème, nous avons proposé un mécanisme de traçage générique et portable qui permet de récupérer des événements spécifiques à la TM afin de mieux analyser les performances des applications. Enfin, nous avons appliqué notre mécanisme de traçage sur deux applications du *benchmark* STAMP et analysé les résultats obtenus.

Ensuite, nous avons adressé le problème de l'amélioration des performances des applications utilisant de la TM sur des plates-formes multicœurs (Chapitre 4). Nous avons souligné que le placement des *threads* (*thread mapping*) était très important et pouvait améliorer considérablement les performances globales obtenues. Nous nous sommes intéressés à récupérer des informations utiles à partir des applications et des systèmes de STM et ensuite utiliser ces informations pour mieux exploiter la hiérarchie mémoire des plates-formes multicœurs modernes.

Pour faire face à la grande diversité des applications, des systèmes de STM et des plates-formes, nous avons proposé une approche basée sur l'Apprentissage Automatique (*Machine Learning* – ML) pour prédire automatiquement les stratégies de placement de *threads* les plus appropriées pour des applications basées sur de la TM. Nous avons mis en œuvre deux approches pour effectuer le placement de *threads* pour des applications à base de TM dans TinySTM d'une façon statique et dynamique.

Finalement, nous avons évalué les performances obtenues par nos deux approches sur des applications de TM (Chapter 5). Nous avons montré que l'approche statique est suffisamment précise et améliore les performances d'un ensemble d'applications d'au maximum 18%. Dans le cas de l'approche dynamique, avons obtenu des améliorations de performances d'au maximum 31% par rapport à la meilleure stratégie statique. Dans l'ensemble, l'amélioration de la performance moyenne de l'approche dynamique par rapport à la meilleure stratégie statique sur chaque application était d'environ 14% (48 sur 56 charges de travail ont présenté de meilleures performances avec l'approche dynamique) et la perte de performance moyenne était d'environ 4% (8 sur 56 charges de travail ont présenté de plus mauvaises performances avec l'approche dynamique).

6.2 Travaux futurs

Les travaux de recherche présentés dans cette thèse peuvent être poursuivis dans plusieurs directions. Ci-dessous, nous détaillons quelques-unes de ces possibilités :

Explorer de nouvelles architectures, systèmes de STM et applications basées sur de la TM. Pendant cette thèse, de nombreux efforts ont été fait tendant à une plus large adoption de la Mémoire Transactionnelle. Des efforts récents sont par exemple l'ajout du support de la TM par la dernière version de la GNU Compiler Collection (GCC 4.7) ainsi que l'ajout d'un support matériel de la TM par les nouveaux processeurs, comme par exemple le BlueGene/Q d'IBM et les *Transactional Synchronization Extensions (TSX)* d'Intel qui seront disponibles dans la future architecture "Haswell". Nous croyons que ces efforts permettront d'élargir l'utilisation de la TM et contribueront également à l'apparition de nouvelles applications réelles basées sur de la TM. De plus, il est prévu que les nouvelles architectures appliqueront aussi une conception NUMA. Un des axes possibles de recherche à long terme est donc l'adaptation de notre prédicteur pour ces nouvelles architectures et applications. Nous pensons qu'il peut être possible d'étendre notre approche pour qu'elle soit utilisée pour des plates-formes NUMA. Cependant, la façon dont les données sont allouées/réparties entre les bancs de mémoire influe également sur les performances globales des applications sur ces plates-formes. Il sera donc nécessaire de considérer non seulement les stratégies de placement des *threads* mais aussi les politiques d'allo-

cation de mémoire pour mieux utiliser les bancs de mémoire NUMA. Nos travaux antérieurs sur l'impact des politiques d'allocation de mémoire sur des plates-formes NUMA [Cas+09 ; Rib+09 ; Rib+12] peuvent être utilisés comme point de départ.

Créer de nouveaux prédicteurs basés sur d'autres algorithmes d'apprentissage automatique. Notre prédicteur s'appuie sur un algorithme d'apprentissage du type arbre de décision (ID3). Dans notre approche, nous avons utilisé le profil de plusieurs applications et systèmes de STM en tant que données d'entrée pour l'algorithme d'apprentissage ID3. Une fois cette phase d'apprentissage terminée, nous avons toujours utilisé le même prédicteur pour toutes les nouvelles applications inconnues. Ce processus est connu sous le nom *apprentissage automatique hors ligne*. Une autre direction pour des travaux futurs peut être l'utilisation des *algorithmes d'apprentissage automatique en ligne*. Dans l'apprentissage en ligne, les informations provenant des nouvelles applications profilées peuvent être utilisées pour ajuster automatiquement le prédicteur. Nous croyons que cette approche peut être utile pour améliorer la précision de la prédiction sur un plus grand nombre d'applications à base de TM.

Étendre le prédicteur pour qu'il utilise d'autres mesures du système. Dans cette thèse, nous avons utilisé un ensemble de critères qui devront être profilés, tels que l'*abort ratio*, le défaut de cache et les politiques de détection et de résolution de conflits. Une autre mesure intéressante qui pourrait être rajoutée est la consommation d'énergie. L'énergie est de plus en plus l'une des ressources les plus chères et l'élément de coût le plus important pour faire fonctionner des plates-formes dotées d'un grand nombre de cœurs. Un scénario réaliste est donc qu'un utilisateur veuille utiliser la consommation d'énergie au lieu du temps d'exécution en tant qu'indicateur de performance dans le prédicteur. Dans ce cas, le prédicteur choisirait une stratégie de placement de *threads* qui consomme moins d'énergie.

Bibliography

[Alb+05] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. "BioBench: A Benchmark Suite of Bioinformatics Applications". In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Austin, USA: IEEE Computer Society, 2005, pp. 2–9. ISBN: 0-7803-8965-4. DOI: 10.1109/ISPASS.2005.1430554.

[Ans+08] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. "Lee-TM: A Non-trivial Benchmark for Transactional Memory". In: *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*. Vol. 5022. Lecture Notes in Computer Science (LNCS). Aiya Napa, Cyprus: Springer-Verlag, 2008, pp. 196–207. ISBN: 978-3-540-69500-4. DOI: 10.1007/978-3-540-69501-1_21.

[Ans+09] Mohammad Ansari, Kim Jarvis, Christos Kotselidis, Mikel Luján, Chris Kirkham, and Ian Watson. "Profiling Transactional Memory Applications". In: *International Conference on Parallel, Distributed, and Network-based Processing (PDP)*. Weimar, Germany: IEEE Computer Society, 2009, pp. 11–20. ISBN: 978-0-7695-3544-9. DOI: 10.1109/PDP.2009.35.

[App+07] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton Series in Applied Mathematics. Princeton, USA: Princeton University Press, 2007.

[Asa+09] Krste Asanovic et al. “A View of the Parallel Computing Landscape”. In: *Communications of the ACM* 52 (10 2009), pp. 56–67. DOI: 10.1145/1562764.1562783.

[Bai+95] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. *The NAS Parallel Benchmarks 2.0*. Tech. rep. NAS-95-020. NASA Ames Research Center, 1995. URL: http://www.nas.nasa.gov/assets/pdf/techreports/1995/nas-95-020.pdf.

[Bou+10] Remco R. Bouckaert, Eibe Frank, Mark A. Hall, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. “WEKA: Experiences with a Java Open-Source Project”. In: *Journal of Machine Learning Research* 11 (2010), pp. 2533–2541.

[Bro+10] François Broquedis, Jérôme Clet-Ortega, Stephanie Moreaud, Brice Goglin, Guillaume Mercier, and Samuel Thibault. “hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications”. In: *Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)*. Pisa, Italy: IEEE Computer Society, 2010, pp. 180–186. ISBN: 978-1-4244-5672-7. DOI: 10.1109/PDP.2010.67.

[Cas+08] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. “Software Transactional Memory: Why is it Only a Research Toy?” In: *Communications of the ACM* 51.11 (2008), pp. 40–46. ISSN: 0001-0782. DOI: 10.1145/1400214.1400228.

[Cas+09] Márcio Castro, Luiz Gustavo Fernandes, Christiane Pousa Ribeiro, Jean-François Méhaut, and Marilton S. de Aguiar. “NUMA-ICTM: A Parallel Version of ICTM Exploiting Memory Placement Strategies for NUMA Machines”. In: *International Parallel and Distributed Processing Symposium (IPDPS)*. Rome, Italy: IEEE Computer Society, 2009, pp. 1–8. ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5161155.

[Cas+10] Márcio Castro, Kiril Georgiev, Vania Marangonzova-Martin, Jean-François Méhaut, Luiz Gustavo Fernandes, and Miguel Santana. *Analyzing Soft-*

ware Transactional Memory Applications by Tracing Transactions. Tech. rep. RR-7334. INRIA, 2010.

[Cas+11a] Márcio Castro, Kiril Georgiev, Vania Marangonzova-Martin, Jean-François Méhaut, Luiz Gustavo Fernandes, and Miguel Santana. "Analysis and Tracing of Applications Based on Software Transactional Memory on Multicore Architectures". In: *Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)*. Aya Napa, Cyprus: IEEE Computer Society, 2011, pp. 199–206. ISBN: 978-0-7695-4328-4. DOI: 10.1109/PDP.2011.27.

[Cas+11b] Márcio Castro, Luís Fabricio Wanderley Góes, Christiane Pousa Ribeiro, Murray Cole, Marcelo Cintra, and Jean-François Méhaut. "A Machine Learning-Based Approach for Thread Mapping on Transactional Memory Applications". In: *High Performance Computing Conference (HiPC)*. Bangalore, India: IEEE Computer Society, 2011, pp. 1–10. ISBN: 978-1-4577-1949-3. DOI: 10.1109/HiPC.2011.6152736.

[Cas+12a] Márcio Castro, Luís Fabrício Góes, Luiz Gustavo Fernandes, and Jean-François Méhaut. "Dynamic Thread Mapping Based on Machine Learning for Transactional Memory Applications". In: *Euro-TM Workshop on Transactional Memory (WTM)*. Extended abstract. Apr. 2012.

[Cas+12b] Márcio Castro, Luís Fabrício Góes, Luiz Gustavo Fernandes, and Jean-François Méhaut. "Dynamic Thread Mapping Based on Machine Learning for Transactional Memory Applications". In: *International European Conference on Parallel and Distributed Computing (Euro-Par)*. Vol. 7484. Lecture Notes in Computer Science (LNCS). Rhodes Island, Greece: Springer-Verlag, 2012, pp. 465–476. ISBN: 978-3-642-32819-0. DOI: 10.1007/978-3-642-32820-6_47.

[Cas10] Márcio Castro. *Software Transactional Memory on Parallel Programming Environments*. Tech. rep. LIG Laboratory, Nano2012-OPM2 Project, 2010.

[CD10] Márcio Castro and Augustin Degomme. *Transactional Memory: State of Art and Trends*. Tech. rep. LIG Laboratory, Nano2012-OPM2 Project, 2010.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. "The 007 Benchmark". In: *ACM SIGMOD Record* 22.2 (1993), pp. 12–31. DOI: 10.1145/170036.170041.

[Che+06] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and Harold Kuhn. "MPIPP: An Automatic Profile-Guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters". In: *International Conference on Supercomputing (ICS)*. Cairns, Australia: ACM, 2006, pp. 353–360. ISBN: 1-59593-282-8. DOI: 10.1145/1183401.1183451.

[Chu+06] Jaewoong Chung, Hassan Chafi, Chi Cao Minh, Austen Mcdonald, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. "The Common Case Transactional Behavior of Multithreaded Programs". In: *International Symposium on High-Performance Computer Architecture (HPCA)*. Austin, USA: IEEE Computer Society, 2006.

[Cru+11] Eduardo Henrique Molina da Cruz, Marco Antonio Zanata Alves, Alexandre Carissimi, Philippe Olivier Alexandre Navaux, Christiane Pousa Ribeiro, and Jean-François Méhaut. "Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms". In: *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. IEEE Computer Society, 2011, pp. 551–558. ISBN: 978-1-61284-425-1. DOI: 10.1109/IPDPS.2011.197.

[CS00] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. 1st. Cambridge University Press, 2000. ISBN: 978-0521780193.

[Cza11] Ireneusz Czarnowski. "Distributed Learning with Data Reduction". In: *Transactions on Computational Collective Intelligence IV*. Vol. 6660. Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2011, pp. 3–121. ISBN: 978-3-642-21883-5. DOI: 10.1007/978-3-642-21884-2.

[Dal+10] Luke Dalessandro, Dave Dice, Michael Scott, Nir Shavit, and Michael Spear. “Transactional Mutex Locks”. In: *International European Conference on Parallel and Distributed Computing (Euro-Par)*. Vol. 6272. Lecture Notes in Computer Science (LNCS). Ischia, Italy: Springer-Verlag, 2010, pp. 2–13. ISBN: 3-642-15290-2. DOI: 10.1007/978-3-642-15291-7.

[Dam+06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. “Hybrid Transactional Memory”. In: *SIGPLAN Notices* 41.11 (Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems 2006), pp. 336–346. DOI: 10.1145/1168918.1168900.

[DD06] Mathieu Desnoyers and Michel R. Dagenais. “The LTTng Tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux”. In: *Linux Symposium*. Ottawa, Canada, 2006.

[DGK09] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. “Stretching Transactional Memory”. In: *ACM SIGPLAN Notices* 44.6 (Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation 2009), pp. 155–165. ISSN: 0362-1340. DOI: 10.1145/1543135.1542494.

[Dic+09] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. “Early Experience with a Commercial Hardware Transactional Memory Implementation”. In: *SIGPLAN Notices* 44.3 (Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems 2009), pp. 157–168. DOI: 10.1145/1508244.1508263.

[Die+10] Matthias Diener, Felipe L. Madruga, Eduardo R. Rodrigues, Marco Antonio Z. Alves, Joerg Schneider, Philippe Olivier A. Navaux, and Hans-Ulrich Heiss. “Evaluating Thread Placement Based on Memory Access Patterns for Multi-core Processors”. In: *IEEE International Conference on High Performance Computing and Communications (HPCC)*. Melbourne, Australia: IEEE Computer Society, 2010, pp. 491–496. ISBN: 978-1-4244-8335-8. DOI: 10.1109/HPCC.2010.114.

[Dra+11] Aleksandar Dragojević, Pascal Felber, Vicent Gramoli, and Rachid Guerraoui. “Why STM Can be More Than a Research Toy”. In: *Communications of the ACM* 54.4 (2011), pp. 70–77. DOI: 10.1145/1924421.1924440.

[DS06] Dave Dice and Nir Shavit. “What Really Makes Transactions Faster?” In: *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*. Ottawa, Canada: ACM, 2006.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. “Transactional Locking II”. In: *International Symposium on Distributed Computing (DISC)*. Vol. 4167. Lecture Notes in Computer Science (LNCS). Stockholm, Sweden: Springer-Verlag, 2006, pp. 194–208. ISBN: 978-3-540-44624-8. DOI: 10.1007/11864219_14.

[Fel+07] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. “Transactifying Applications using an Open Compiler Framework”. In: *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*. Portland, USA: ACM, 2007.

[FFR08] Pascal Felber, Christof Fetzer, and Torvald Riegel. “Dynamic Performance Tuning of Word-Based Software Transactional Memory”. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Salt Lake City, USA: ACM, 2008, pp. 237–246. ISBN: 978-1-59593-795-7. DOI: 10.1145/1345206.1345241.

[GCC12] GCC. *GNU Compiler Collection*. 2012. URL: http://gcc.gnu.org.

[GHC12] GHC. *Glasgow Haskell Compiler*. 2012. URL: http://www.haskell.org/ghc.

[GHP05] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. “Toward a Theory of Transactional Contention Managers”. In: *ACM Symposium on Principles of Distributed Computing (PODC)*. Las Vegas, USA: ACM, 2005, pp. 258–264. ISBN: 1-59593-994-2. DOI: 10.1145/1073814.1073863.

[GKV07] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. “STMBench7: A Benchmark for Software Transactional Memory”. In: *ACM SIGOPS Operating Systems Review* 41.3 (Proceedings of the European Conference on Computer Systems Mar. 2007), pp. 315–324. DOI: 10.1145/1272998.1273029.

[GO11] Dominik Grewe and Michael F. P. O’Boyle. “A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL”. In: *International Conference on Compiler Construction (CC)*. Saarbrücken, Germany: Springer-Verlag, 2011, pp. 286–305. ISBN: 978-3-642-19860-1.

[GUW08] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Wi. *Database Systems: The Complete Book*. 2nd. Prentice Hall, 2008. ISBN: 978-0-131-87325-4.

[HBK06] Jim Held, Jerry Bautista, and Sean Koehl. *From a Few Cores to Many: A Tera-scale Computing Research Overview Editors*. Tech. rep. Intel Whitepaper. California, USA: Intel Corporation, 2006.

[HLM06] Maurice Herlihy, Victor Luchangco, and Mark Moir. “A Flexible Framework for Implementing Software Transactional Memory”. In: *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Portland, USA: ACM, 2006. ISBN: 1-59593-348-4. DOI: 10.1145/1167473.1167495.

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory: Synthesis Lectures on Computer Architecture*. 2nd. Vol. 5. 1. Madison, USA: Morgan & Claypool Publishers, 2010, p. 263. DOI: 10.2200/S00272ED1V01Y201006CAC011.

[HM93] Maurice Herlihy and J. Eliot B. Moss. “Transactional Memory: Architectural Support for Lock-Free Data Structures”. In: *International Symposium on Computer Architecture (ISCA)*. San Diego, USA: IEEE Computer Society, 1993, pp. 289–300. ISBN: 0-8186-3810-9. DOI: 10.1109/ISCA.1993.698569.

[Hon+09] Shengyan Hong, Sri H. K. Narayanan, Mahmut T. Kandemir, and Özcan Özturk. “Process Variation Aware Thread Mapping for Chip Multiprocessors”. In: *Design, Automation & Test in Europe (DATE)*. Nice, France: IEEE Computer Society, 2009, pp. 821–826. ISBN: 978-3-9810-8015-5.

[Hon+10] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. “EigenBench: A Simple Exploration Tool for Orthogonal TM Characteristics”. In: *IEEE International Symposium on Workload Characterization (IISWC)*. Atlanta, USA: IEEE Computer Society, 2010, pp. 1–11. ISBN: 978-1-4244-9297-8. DOI: 10.1109/IISWC.2010.5648812.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Burlington, USA: Morgan Kaufmann, Mar. 2008. ISBN: 978-0-12-370591-4.

[HX98] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. 1st. McGraw-Hill, 1998. ISBN: 978-0-07031-798-7.

[HYH12] Zhengyu He, Xiao Yu, and Bo Hong. “Profiling-based Adaptive Contention Management for Software Transactional Memory”. In: *International Parallel and Distributed Processing Symposium (IPDPS)*. Shanghai, China: IEEE Computer Society, 2012, pp. 1204–1215. DOI: 10.1109/IPDPS.2012.110.

[Int09] Intel. *Intel C++ STM Compiler Prototype Edition 3.0 User’s Guide*. 2009. URL: http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/.

[JK81] Xiong Ji-Guang and Tokinori Kozawa. “An Algorithm for Searching Shortest Path by Propagating Wave Fronts in Four Quadrants”. In: *Design Automation Conference (DAC)*. Nashville, Tennessee, United States: IEEE Press, 1981, pp. 29–36.

[KB12] Laxmikant V. Kale and Abhinav Bhatele, eds. *Parallel Science and Engineering Applications: The Charm++ Approach*. 1st. CRC Press, 2012. ISBN: 1466504129.

[KCP06] Milind Kulkarni, L. Paul Chew, and Keshav Pingali. “Using Transactions in Delaunay Mesh Generation”. In: *Workshop on Transactional Memory Workloads*. Ottawa, Canada: ACM, 2006, pp. 23–31.

[Kes+09] Gokcen Kestor, Srdjan Stipic, Osman Unsal, Adrián Cristal, and Mateo Valero. “RMS-TM: A Transactional Memory Benchmark For Recognition, Mining And Synthesis Applications”. In: *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*. Raleigh, USA: ACM, 2009.

[Kni86] Thomas F. Knight. “An Architecture for Mostly Functional Languages”. In: *ACM Lisp and Functional Programming Conference*. Cambridge, USA: ACM, Aug. 1986, pp. 500–519. ISBN: 0-89791-200-4. DOI: 10.1145/319838.319854.

[Knu+08] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. “The Vampir Performance Analysis Tool-Set”. In: *Tools for High Performance Computing* (2008), pp. 139–155. DOI: 10.1007/978-3-540-68564-7_9.

[Kum+06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. “Hybrid Transactional Memory”. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. New York, USA: ACM, 2006. ISBN: 1-59593-189-9. DOI: 10.1145/1122971.1123003.

[LK08] James Larus and Christos Kozyrakis. “Transactional Memory: Is TM the Answer for Improving Parallel Programming?” In: *Communications of ACM* 51.7 (2008), pp. 80–88. ISSN: 0001-0782. DOI: 10.1145/1364782.1364800.

[Lom77] David Bruce Lomet. “Process Structuring, Synchronization, and Recovery Using Atomic Actions”. In: *SIGPLAN Notices* 12.3 (Proceedings of the ACM Conference on Language Design for Reliable Software 1977), pp. 128–137. DOI: 10.1145/390017.808319.

[Lou+09] João Lourenço, Ricardo Dias, João Luis, Miguel Rebelo, and Vasco Pessanha. “Understanding the Behavior of Transactional Memory Applications”. In: *ACM Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*. Chicago, USA: ACM, 2009, pp. 31–39. ISBN: 978-1-60558-655-7. DOI: 10.1145/1639622.1639625.

[Mal+11] Walther Maldonado, Patrick Marlier, Pascal Felber, Julia Lawall, Giller Muller, and Etienne Rivière. “Deadline-Aware Scheduling for Software Transactional Memory”. In: *International Conference on Dependable Systems & Networks (DSN)*. Hong Kong, China: IEEE Computer Society, 2011, pp. 257–268. ISBN: 978-1-4244-9231-2. DOI: 10.1109/DSN.2011.5958224.

[Mar+06] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer, and Michael L. Scott. “Lowering the Overhead of Software Transactional Memory”. In: *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*. Ottawa, Canada: ACM, 2006.

[McD+05] Austen McDonald, JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Brian D. Carlstrom, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. “Characterization of TCC on Chip-Multiprocessors”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Saint Louis, USA: IEEE Computer Society, 2005, pp. 63–74. ISBN: 0-7695-2429-X. DOI: 10.1109/PACT.2005.11.

[McK04] Sally A. McKee. “Reflections on the Memory Wall”. In: *Conference on Computing Frontiers*. Ischia, Italy: ACM, 2004, pp. 162–168. ISBN: 1-58113-741-9. DOI: 10.1145/977091.977115.

[Min+08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. “STAMP: Stanford Transactional Applications for Multi-Processing”. In: *IEEE International Symposium on Workload Characterization (IISWC)*. Seattle, USA: IEEE Computer Society, 2008, pp. 35–46. ISBN: 978-1-4244-2777-2. DOI: 10.1109/IISWC.2008.4636089.

[MIS05] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. "Adaptive Software Transactional Memory". In: *International Symposium on Distributed Computing (DISC)*. Vol. 3724. LNCS. Krakow, Poland: Springer-Verlag, 2005, pp. 354–368. ISBN: 3-540-29163-6.

[Mit97] Tom M. Mitchell. *Machine Learning*. 1st. New York, USA: McGraw-Hill, 1997. ISBN: 978-0070428072.

[ML06] Milo Martin and Colin Blundelland E. Lewis. "Subtleties of Transactional Memory Atomicity Semantics". In: *IEEE Computer Architecture Letters* 5 (2 2006). DOI: 10.1109/L-CA.2006.18.

[MM08] Virendra Jayant Marathe and Mark Moir. "Toward High Performance Nonblocking Software Transactional Memory". In: *PPoPP '08: Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Salt Lake City, UT, USA: ACM, 2008, pp. 227–236.

[Moo+06] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. "LogTM: Log-based Transactional Memory". In: *International Symposium on High-Performance Computer Architecture (HPCA)*. Austin, USA: IEEE Computer Society, 2006, pp. 254–265. ISBN: 0-7803-9368-6. DOI: 10.1109/HPCA.2006.1598134.

[MSS08] Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. "Scalable Techniques for Transparent Privatization in Software Transactional Memory". In: *International Conference on Parallel Processing (ICPP)*. Portland, USA: IEEE Computer Society, 2008, pp. 67–74. ISBN: 978-0-7695-3374-2. DOI: 10.1109/ICPP.2008.69.

[Nar+06] Ramanathan Narayanan, Berkin Ö. Ikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. "MineBench: A Benchmark Suite for Data Mining Workloads". In: *IEEE International Symposium on Workload Characterization*. San Jose, USA: IEEE Computer Society, 2006, pp. 182–188. ISBN: 1-4244-0508-4. DOI: 10.1109/IISWC.2006.302743.

[Oli+11] Poliana Oliveira, Henrique Cota de Freitas, Christiane Pousa Ribeiro, Márcio Castro, Vania Marangonzova-Martin, and Jean-François Méhaut. "Performance Evaluation of WiNoCs for Parallel Workloads Based on

Collective Communications". In: *IADIS International Conference on Applied Computing (AC)*. Rio de Janeiro, Brazil: IADIS Press, 2011, pp. 307–314. ISBN: 978-989-8533-06-7.

[PK01] Athanassios Papagelis and Dimitrios Kalles. "Breeding Decision Trees Using Evolutionary Techniques". In: *International Conference on Machine Learning (ICML)*. Williamstown, USA: Morgan Kaufmann, 2001, pp. 393–400. ISBN: 1-55860-778-1.

[PW10] Donal E. Porter and Emmett Witchel. "Understanding Transactional Memory Performance". In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. White Plains, EUA: IEEE Computer Society, 2010, pp. 97–108. ISBN: 978-1-4244-6023-6. DOI: 10.1109/ISPASS.2010.5452061.

[Qui08] Michael Jay Quinn. *Parallel Programming in C with MPI and OpenMP*. New York, NY: McGraw-Hill, 2008. ISBN: 0071232656.

[Qui86] John Ross Quinlan. "Induction of Decision Trees". In: *Machine Learning* 1.1 (1986), pp. 81–106. ISSN: 0885-6125. DOI: 10.1023/A:1022643204877.

[Rib+09a] Christiane Pousa Ribeiro, Márcio Castro, Luiz Gustavo Fernandes, Jean-François Méhaut, Alexandre Carissimi, and Fabrice Dupros. "High Performance Applications on Hierarchical Shared Memory Multiprocessors". In: *Colloque d'Informatique: Brésil / INRIA, Coopérations, Avancées et Défis (COLIBRI)*. Bento Gonçalves, Brazil: Brazilian Computer Society, 2009, pp. 55–60.

[Rib+09b] Christiane Pousa Ribeiro, Jean-François Méhaut, Alexandre Carissimi, Márcio Castro, and Luiz Gustavo Fernandes. "Memory Affinity for Hierarchical Shared Memory Multiprocessors". In: *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. São Paulo, Brazil: IEEE Computer Society, 2009, pp. 59–66. ISBN: 978-0-7695-3857-0. DOI: 10.1109/SBAC-PAD.2009.16.

[Rib+10] Christiane Pousa Ribeiro, Márcio Castro, Jean-François Méhaut, and Alexandre Carissimi. “Improving Memory Affinity of Geophysics Applications on NUMA Platforms Using Minas”. In: *International Meeting on High Performance Computing for Computation Science (VECPAR)*. Vol. 6449. Lecture Notes in Computer Science (LNCS). Berkeley, USA: Springer-Verlag, 2010, pp. 272–292. ISBN: 978-3-642-19327-9. DOI: 10.1007/978-3-642-19328-6_27.

[Rib+11] Christiane Pousa Ribeiro, Márcio Castro, Jean-François Méhaut, Vania Marangonzova-Martin, Henrique Cota de Freitas, and Carlos Augusto Paiva da Silva Martins. “Investigating the Impact of CPU and Memory Affinity on Multi-core Platforms: A Case Study of Numerical Scientific Multithreaded Applications”. In: *IADIS International Conference on Applied Computing (AC)*. Rio de Janeiro, Brazil: IADIS Press, 2011, pp. 299–306. ISBN: 978-989-8533-06-7.

[Rib+12] Christiane Pousa Ribeiro, Márcio Castro, Vania Marangonzova-Martin, Jean-François Méhaut, Henrique Cota de Freitas, and Carlos Augusto Paiva da Silva Martins. “Evaluating CPU and Memory Affinity for Numerical Scientific Multithreaded Benchmarks on Multi-cores”. In: *IADIS International Journal on Computer Science and Information Systems (IJCSIS)* (1 2012). ISSN: 1646-3692.

[Rib11] Christiane Pousa Ribeiro. “Contributions on Memory Affinity Management for Hierarchical Shared Memory Multi-core Platforms”. PhD thesis. Université de Grenoble, June 2011.

[Ros+07] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Aditya Bhandari, and Emmett Witchel. “TxLinux: Using and Managing Hardware Transactional Memory in an Operating System”. In: *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Stevenson, USA: ACM, 2007, pp. 87–102. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294271.

[SBS11] S. Subramanya Sastry, Rastislav Bodík, and James E. Smith. “Rapid Profiling via Stratified Sampling”. In: *ACM SIGARCH Computer Architecture*

News 29.2 (Proceedings of the International Symposium on Computer Architecture 2011), pp. 278–289. DOI: 10.1145/384285.379273.

[SH11] Robert Schöne and Daniel Hackenberg. "On-Line Analysis of Hardware Performance Events for Workload Characterization and Processor Frequency Scaling Decisions". In: *International Conference on Performance Engineering (ICPE)*. Karlsruhe, Germany: ACM, 2011, pp. 481–486. ISBN: 978-1-4503-0519-8. DOI: 10.1145/1958746.1958819.

[She99] Sameer Shende. "Profiling and Tracing in Linux". In: *USENIX Annual Technical Conference*. Monterey, USA: USENIX Association, 1999.

[Shr+06] Arrvindh Shriraman, Virendra J. Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N. Scherer III, and Michael F. Spear. "Hardware Acceleration of Software Transactional Memory". In: *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*. Ottawa, Canada: ACM, 2006. ISBN: 1-59593-320-4.

[SM06] Sameer S. Shende and Allen D. Malony. "The TAU Parallel Performance System". In: *International Journal of High Performance Computing Applications* 20 (2 May 2006), pp. 287–311. DOI: 10.1177/1094342006064482.

[Tai94] Kuo-Chung Tai. "Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs". In: *International Conference on Parallel Processing (ICPP)*. Vol. 2. Raleigh, USA: IEEE Computer Society, 1994, pp. 69–72. ISBN: 0-8493-2494-7. DOI: 10.1109/ICPP.1994.84.

[Ter+10] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. "Collecting Performance Data with PAPI-C". In: *Tools for High Performance Computing*. Springer-Verlag, 2010, pp. 157–173. DOI: 10.1007/978-3-642-11261-4_11.

[Tou+09] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. "Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping". In: *ACM SIGPLAN Notices* 44.6 (Proceedings of the ACM

SIGPLAN Conference on Programming Language Design and Implementation 2009), pp. 177–187. DOI: 10.1145/1543135.1542496.

[VIR07] VIRTUTECH SIMICS. *Simics 3.0 – User Guide for Unix*. 2007. URL: http://www.simics.net.

[Wam+12] Jons-Tobias Wamhoff, Christof Fetzer, Pascal Felber, Etienne Rivière, and Gilles Muller. "FastLane: Streamlining Transactions for Low Thread Counts". In: *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*. New Orleans, USA: ACM, 2012.

[Wan+11] Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. "Towards Applying Machine Learning to Adaptive Transactional Memory". In: *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*. San Jose, USA: ACM, 2011.

[Wan09] Ruibo Wang. "Investigating Software Transactional Memory on Big SMP Machines". In: *ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD)*. Daegu, Korea: IEEE Computer Society, 2009, pp. 507–509. ISBN: 978-0-7695-3642-2. DOI: 10.1109/SNPD.2009.25.

[WL10] Ruibo Wang and Kai Lu. "Using Transactional Memory on CC-NUMA Systems". In: *International Conference on Networked Computing (INC)*. Gyeongju, South Korea: IEEE Computer Society, 2010, pp. 1–3. ISBN: 978-89-88678-20-6.

[WLL09] Ruibo Wang, Kai Lu, and Xicheng Lu. "Investigating Transactional Memory Performance on ccNUMA Machines". In: *International Symposium on High Performance Distributed Computing (HPDC)*. Garching, Germany: ACM, 2009, pp. 67–68. ISBN: 978-1-60558-587-1. DOI: 10.1145/1551609.1551625.

[WO09] Zheng Wang and Michael F.P. O'Boyle. "Mapping Parallelism to Multicores: A Machine Learning Based Approach". In: *ACM SIGPLAN Notices* 44.4 (Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2009), pp. 75–84. DOI: 10.1145/1594835.1504189.

[Yen+07] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. "LogTM-SE: Decoupling Hardware Transactional Memory from Caches". In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Phoenix, USA: IEEE Computer Society, 2007, pp. 261–272. ISBN: 1-4244-0804-0. DOI: 10.1109/HPCA.2007.346204.

[Zha+09] Jin Zhang, Jidong Zhai, Wenguang Chen, and Weimin Zheng. "Process Mapping for MPI Collective Communications". In: *International European Conference on Parallel and Distributed Computing (Euro-Par)*. Vol. 5704. Lecture Notes in Computer Science (LNCS). Ischia, Italy: Springer-Verlag, 2009, pp. 81–92. ISBN: 978-3-642-03868-6. DOI: 10.1007/978-3-642-03869-3_11.

[Zha00] Guoqiang Peter Zhang. "Neural Networks for Classification: A Survey". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews* 30.4 (2000), pp. 451–462. DOI: 10.1109/5326.897072.

[Zyu+08] Ferad Zyulkyarov, Adrian Cristal, Sanja Cvijic, Eduard Ayguade, Mateo Valero, Osman Unsal, and Tim Harris. "WormBench: A Configurable Workload for Evaluating Transactional Memory Systems". In: *Workshop on Memory Performance: Dealing with Applications, Systems and Architecture (MEDEA)*. Toronto, Canada: ACM, 2008, pp. 61–68. ISBN: 978-1-60558-243-6. DOI: 10.1145/1509084.1509093.

[Zyu+10] Ferad Zyulkyarov, Srdjan Stipic, Tim Harris, Osman Unsal, Adrián Cristal, Ibrahim Hur, and Mateo Valero. "Discovering and Understanding Performance Bottlenecks in Transactional Applications". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Vienna, Austria: ACM, 2010, pp. 285–294. ISBN: 978-1-4503-0178-7. DOI: 10.1145/1854273.1854311.

Abstract

Multicore processors are now a mainstream approach to deliver higher performance to parallel applications. In order to develop efficient parallel applications for those platforms, developers must take care of several aspects, ranging from the architectural to the application level. In this context, Transactional Memory (TM) appears as a programmer friendly alternative to traditional lock-based concurrency for those platforms. It allows programmers to write parallel code as transactions, which are guaranteed to execute atomically and in isolation regardless of eventual data races. At runtime, transactions are executed speculatively and conflicts are solved by re-executing conflicting transactions. Although TM intends to simplify concurrent programming, the best performance can only be obtained if the underlying runtime system matches the application and platform characteristics.

The contributions of this thesis concern the analysis and improvement of the performance of TM applications based on Software Transactional Memory (STM) on multicore platforms. Firstly, we show that the TM model makes the performance analysis of TM applications a daunting task. To tackle this problem, we propose a generic and portable tracing mechanism that gathers specific TM events, allowing us to better understand the performances obtained. The traced data can be used, for instance, to discover if the TM application presents points of contention or if the contention is spread out over the whole execution. Our tracing mechanism can be used with different TM applications and STM systems without any changes in their original source codes.

Secondly, we address the performance improvement of TM applications on multicores. We point out that thread mapping is very important for TM applications and it can considerably improve the global performances achieved. To deal with the large diversity of TM applications, STM systems and multicore platforms, we propose an approach based on Machine Learning to automatically predict suitable thread mapping strategies for TM applications. During a prior learning phase, we profile several TM applications running on different STM systems to construct a predictor. We then use the predictor to perform static or dynamic thread mapping in a state-of-the-art STM system, making it transparent to the users.

Finally, we perform an experimental evaluation and we show that the static approach is fairly accurate and can improve the performance of a set of TM applications by up to 18%. Concerning the dynamic approach, we show that it can detect different phase changes during the execution of TM applications composed of diverse workloads, predicting thread mappings adapted for each phase. On those applications, we achieve performance improvements of up to 31% in comparison to the best static strategy.

Résumé

Le concept de processeur multicœurs constitue le facteur dominant pour offrir des hautes performances aux applications parallèles. Afin de développer des applications parallèles capables de tirer profit de ces plate-formes, les développeurs doivent prendre en compte plusieurs aspects, allant de l'architecture aux caractéristiques propres à l'application. Dans ce contexte, la Mémoire Transactionnelle (*Transactional Memory* – TM) apparaît comme une alternative intéressante à la synchronisation basée sur les verrous pour ces plates-formes. Elle permet aux programmeurs d'écrire du code parallèle encapsulé dans des transactions, offrant des garanties comme l'atomicité et l'isolement. Lors de l'exécution, les opérations sont exécutées spéculativement et les conflits sont résolus par ré-exécution des transactions en conflit. Bien que le modèle de TM ait pour but de simplifier la programmation concurrente, les meilleures performances ne pourront être obtenues que si l'exécutif est capable de s'adapter aux caractéristiques des applications et de la plate-forme.

Les contributions de cette thèse concernent l'analyse et l'amélioration des performances des applications basées sur la Mémoire Transactionnelle Logicielle (*Software Transactional Memory* – STM) pour des plates-formes multicœurs. Dans un premier temps, nous montrons que le modèle de TM et ses performances sont difficiles à analyser. Pour s'attaquer à ce problème, nous proposons un mécanisme de traçage générique et portable qui permet de récupérer des événements spécifiques à la TM afin de mieux analyser les performances des applications. Par exemple, les données tracées peuvent être utilisées pour détecter si l'application présente des points de contention ou si cette contention est répartie sur toute l'exécution. Notre approche peut être utilisée sur différentes applications et systèmes STM sans modifier leurs codes sources.

Ensuite, nous abordons l'amélioration des performances des applications sur des plate-formes multicœurs. Nous soulignons que le placement des threads (*thread mapping*) est très important et peut améliorer considérablement les performances globales obtenues. Pour faire face à la grande diversité des applications, des systèmes STM et des plates-formes, nous proposons une approche basée sur l'Apprentissage Automatique (*Machine Learning*) pour prédire automatiquement les stratégies de placement de threads appropriées pour les applications de TM. Au cours d'une phase d'apprentissage préliminaire, nous construisons les profils des applications s'exécutant sur différents systèmes STM pour obtenir un prédicteur. Nous utilisons ensuite ce prédicteur pour placer les threads de façon statique ou dynamique dans un système STM récent.

Finalement, nous effectuons une évaluation expérimentale et nous montrons que l'approche statique est suffisamment précise et améliore les performances d'un ensemble d'applications d'un maximum de 18%. En ce qui concerne l'approche dynamique, nous montrons que l'on peut détecter des changements de phase d'exécution des applications composées des diverses charges de travail, en prévoyant une stratégie de placement appropriée pour chaque phase. Sur ces applications, nous avons obtenu des améliorations de performances d'un maximum de 31% par rapport à la meilleure stratégie statique.