UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Programa de Pós-Graduação em Computação

Thesis

Sharing-Aware Thread Mapping in Software Transactional Memory

Douglas Pereira Pasqualin

Pelotas, 2021

Douglas Pereira Pasqualin

Sharing-Aware Thread Mapping in Software Transactional Memory

Thesis presented to the Graduate Program in Computing at the Technology Development Center of the Federal University of Pelotas, in partial fulfillment of the requirements for the degree of Doctor of Computer Science.

Advisor: Dr. André Rauber Du Bois
Coadvisores: Dr. Matthias Diener
Dr. Maurício Lima Pilla

Pelotas, 2021

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação na Publicação

P284s Pasqualin, Douglas Pereira

Sharing-aware thread mapping in software transactional memory / Douglas Pereira Pasqualin ; André Rauber Du Bois, orientador ; Matthias Diener, Maurício Lima Pilla, coorientadores. — Pelotas, 2021.

114 f. : il.

Tese (Doutorado) — Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2021.

1. Software transactional memory. 2. Sharing-aware. 3. Thread mapping. 4. Communication. I. Bois, André Rauber Du, orient. II. Diener, Matthias, coorient. III. Pilla, Maurício Lima, coorient. IV. Título.

CDD : 005

Elaborada por Aline Herbstrith Batista CRB: 10/1737

Douglas Pereira Pasqualin

Sharing-Aware Thread Mapping in Software Transactional Memory

Thesis approved in partial fulfillment of the requirements for the degree of Doctor of Computer Science, Graduate Program in Computing, Technology Development Center, Federal University of Pelotas.

Defense Date: 28 April 2021

Examination Board :

Dr. André Rauber Du Bois (advisor)
PhD in Computer Science from the Heriot-Watt University, Scotland.

Dr. Laércio Lima Pilla
PhD in Computer Science from the Federal University of Rio Grande do Sul (UFRGS), Brazil and Université Grenoble Alpes, France.

Dr. Alexandro José Baldassin
PhD in Computer Science from the State University of Campinas (UNICAMP), Brazil

Dr. Gerson Geraldo Homrich Cavalheiro
PhD in Informatique Systèmes et Communications from the Institut National Polytechnique de Grenoble, France.

This is dedicated to my wife Daiane and my son Nicolas, for their endless love and encouragement.

ACKNOWLEDGMENTS

Finally, it is time to write this hard part of the thesis. The acknowledgments! Although it is an optional element of the document, I think that it should be mandatory. It is nearly impossible to imagine developing a PhD thesis alone. Many people work together and backstage, but all are fundamental to reach the final objective. Now it is time to thank them all.

Firstly, I would like to thank my advisor team. Indeed, I do not have just an advisor and a co-advisor, but a team of three advisors. Each one started in a distinct phase of this thesis. So, I thank you, Prof. Dr. André Du Bois, for accepting me as a PhD student. Also, for your support and confidence in me. Prof. Dr. Maurício Pilla, for sharing profound knowledge and guidance during my initial studies on the thesis subject. Finally, Dr. Matthias Diener, I have no words to thank you for all your support. Our contact started with a simple question about your work on Facebook and finished with you being officially my co-advisor. My sincerely *Herzlichen Dank*.

I also would like to thank another team that was always working backstage: my family! Especially my wife Daiane, my son Nicolas and my mom Maria. Thank you for the support and encouragement during these four long years, including giving me the strengths to continue when sometimes I thought that I would not be able to make it! But I did it! This achievement is also yours.

Lastly, I would like to thank my employer (CPD and UFSM) to grant me the privilege of getting a license of my professional activities. Hence, it was possible to dedicate my full time to my PhD studies.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and PROCAD/LEAPaD

“Accessing memory is more like mailing a letter than making a phone call”

(THE ART OF MULTIPROCESSOR PROGRAMMING, PAGE 471)

ABSTRACT

PASQUALIN, Douglas Pereira. **Sharing-Aware Thread Mapping in Software Transactional Memory**. Advisor: André Rauber Du Bois. 2021. 114 f. Thesis (Doctorate in Computer Science) – Technology Development Center, Federal University of Pelotas, Pelotas, 2021.

Software Transactional Memory (STM) is an alternative abstraction for thread synchronization in parallel programming. One advantage is simplicity since it is possible to replace the use of explicit locks with atomic blocks, while the STM runtime is responsible to ensure a consistent execution, for instance, without deadlocks and race conditions. Regarding STM performance, many studies already have been made focusing on reducing the number of transactional aborts and conflicts. However, in current multicore architectures with complex memory hierarchies, it is also important to consider where the memory of a program is allocated and how it is accessed. This thesis proposes the use of a technique called *sharing-aware mapping*, which maps threads to cores and memory pages to NUMA nodes based on their memory access behavior to achieve better performance in STM systems. The first major contribution of this thesis is a mechanism to detect sharing behavior directly inside the STM library by tracking and analyzing how threads perform STM operations. The collected information can be used to perform an optimized mapping of the application's threads to cores in order to improve the efficiency of STM operations. The second contribution of this thesis is the characterization of the sharing behavior of STM applications by using information extracted from the STM runtime, providing information to guide thread mapping based on their sharing behavior. The third contribution is a mechanism to perform sharing-aware thread mapping in STM applications. We first introduce *Static-SharingAware* (SSA), which map threads to cores based on a previous analysis of the sharing behavior of STM applications. Next, we introduce STMap, an online, low overhead mechanism to detect the sharing behavior and perform the mapping directly inside the STM library, by tracking and analyzing how threads perform STM operations during the execution. In experiments with the STAMP benchmark suite and synthetic benchmarks, both mechanisms showed performance gains when compared to the default Linux scheduler.

Keywords: Software Transactional Memory. Sharing-aware. Thread Mapping. Communication.

RESUMO

PASQUALIN, Douglas Pereira. **Mapeamento de *Threads* Baseado em Compartilhamento em Memórias Transacionais em Software**. Orientador: André Rauber Du Bois. 2021. 114 f. Tese (Doutorado em Ciência da Computação) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2021.

Memória Transacional em Software (MTS) é uma abstração para a sincronização de *threads* na programação paralela. Uma de suas vantagens é a simplicidade, pois é possível substituir o uso de bloqueios por blocos atômicos. Além disso, a implementação de MTS é responsável por garantir uma execução consistente, por exemplo, sem *deadlocks* ou condições de corrida. Com relação ao desempenho de MTS, existem muitos estudos focados na redução do número de cancelamentos. Contudo, nas atuais arquiteturas *multicore*, com complexas hierarquias de memória, é também importante considerar onde a memória do programa está alocada e como ela é acessada. Esta tese propõe o uso de uma técnica chamada *mapeamento baseado em compartilhamento* a qual consiste em mapear *threads* para núcleos de processamento e páginas de memória para nós NUMA com base no seu padrão de acesso à memória para melhorar o desempenho de aplicações que utilizam MTS. A primeira contribuição desta tese é um mecanismo para detectar o padrão de acesso à memória em bibliotecas de MTS. Ele consiste em rastrear e analisar como *threads* executam operações de MTS. As informações coletadas podem ser utilizadas para criar um mapeamento otimizado de *threads* para núcleos de processamento, com o objetivo de melhorar a eficiência das operações de MTS. A segunda contribuição é a caracterização do padrão de acesso à memória de aplicações que utilizam MTS, fornecendo informações para guiar um mapeamento de *threads* com base no padrão de compartilhamento da aplicação. A terceira contribuição é um mecanismo para efetuar um mapeamento de *threads* baseado em compartilhamento para aplicações que utilizam MTS. Primeiramente é apresentado *Static-SharingAware* (SSA), que baseado em uma análise prévia do padrão de compartilhamento da aplicação, mapeia *threads* para núcleos de processamento de forma estática. Após, é apresentado STMap, um mecanismo que opera dinamicamente e com baixa sobrecarga, com o objetivo de detectar o padrão de acesso à memoria e efetuar o mapeamento de *threads* durante a execução do programa. Em experimentos com o benchmark STAMP e outras aplicações sintéticas, ambos mecanismos apresentaram ganhos de desempenho quando comparados com o escalonador padrão do Linux.

Palavras-chave: Memória Transacional em Software. Sensibilidade ao compartilhamento. Mapeamento de thread. Comunicação.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ACC	Adaptive Concurrency Control
ATS	Adaptive Transaction Scheduling
AR	Abort Ratio
BAT	Best Alternative Transaction
CAR	Collision Avoidance and Resolution
CB	Contention Bit
CDSM	Communication Detection in Shared Memory
CFS	Completely Fair Scheduler
CI	Contention Intensity
CL	Concurrency Level
CM	Contention Manager
CMP	Chip Multiprocessor
CP	Contention Predictor
CPU	Central Processing Unit
CR	Commit Ratio
DA	Distinct Addresses
DoT	Dominant Thread
DS	Dynamic Serializer
DVFS	Dynamic Voltage and Frequency Scaling
F2C2	Flux-based Feedback-driven Concurrency Control
GC	Garbage Collector
GPU	Graphics Processing Unit
HPC	High Performance Computing
HTM	Hardware Transactional Memory
IBS	Instruction-Based Sampling
kMAF	kernel Memory Affinity Framework

LLC	Last-Level Cache
LO-SER	Low-Overhead Serializing Algorithm
LSA	Lazy Snapshot Algorithm
LUTS	Light-Weight User-Level Transaction Scheduler
MAi	Memory Affinity Interface
MI	Mapping Interval
ML	Machine Learning
MPI	Message Passing Interface
MSE	Mean squared error
NPB	NAS Parallel Benchmark
NUMA	Non-Uniform Memory Access
PEW	Percentage of the Effective Work
PoCC	P-only Concurrency Controller
ProPS	Progressively Pessimistic Scheduling
SC	Saturating Counter
SCA	Speculative Contention Avoidance
SI	Sample Interval
SRP	Success-Reward Policy
SSER+	Stricter Serializability
RS	Read-set
TBB	Threading Building Blocks
TCR	Transaction Commit Ratio
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
TM	Transactional Memory
SOA	Steal-on-Abort
SSA	Static-SharingAware
STAMP	Stanford Transactional Applications for Multi-Processing
STM	Software Transactional Memory
UMA	Uniform Memory Access
VIT	Very Important Transaction
WACC	Weighted Adaptive Concurrency Control
WS	Write-set

CONTENTS

1 INTRODUCTION

At the beginning of the year 2000, multicore processors started to be produced. This decision was taken due to the microarchitectural limitations and, higher power consumption and heat dissipation involved on improving the performance of a single CPU (TRONO, 2015). Since then, the number of cores in a single chip is growing every year. Today, desktops and even cellphone processors are multicore. Besides, servers normally have many processors and each one is multicore.

The simplest architecture for multiprocessors systems are based on a single bus, i.e., one or more processors and memory modules use the same bus for communication. In this architecture, every memory word can be read with the same latency. Hence, they are named UMA (Uniform Memory Access). However, these architectures have a scalability problem as the number of CPUs grow, as all communication needs to pass by the single bus.

One alternative is to replace the single bus with multiple nodes, where each node is a multiprocessor connected directly to a local memory module (GAUD et al., 2015). These architectures are called NUMA (Non-Uniform Memory Access) and are becoming dominant in servers (CALCIU et al., 2017). In NUMA machines, programs have access to the entire memory. In a transparent way, data can be stored in the local node or in a node that belongs to other processor (remote node). Interconnect links between nodes are asymmetric and have different bandwidths (LEPERS; QUÉMA; FEDOROVA, 2015). Hence, the location of the data plays an important role in the performance.

Accessing a remote node implies higher latency, making the access time non-uniform, i.e., depends on the location of the data.

In order to better exploit the parallelism available in these modern architectures, software must be parallel and scalable (GRAHN, 2010). An important issue that arises in parallel programming is thread synchronization, and it is the major cause that prevents the scalability of applications (DAVID; GUERRAOUI; TRIGONAKIS, 2013). Synchronization is necessary when multiple concurrent threads need to access at the same time a shared variable and, at least one thread needs to write to this shared variable. If there is no synchronization, a race condition can happen (NETZER; MILLER, 1992), possibly

leading to non-deterministic results (TRONO, 2015). The block of code that needs to be protected in order to not be accessed at the same time by different processes is called *critical section* (RAYNAL, 2013).

Mutual-exclusion locks are one of the most used abstractions to protect critical sections (FRASER; HARRIS, 2007). However, the semantics of locks is not intuitive. Programmers need to explicitly acquire and release locks, making the source code hard to read and debug (ANTHES, 2014). Besides, if more than one lock was acquired in a critical section, they need to be released in the same order, to avoid *deadlocks* (HERLIHY; SHAVIT, 2008). The performance of locks depends on the size of the critical section that it protects. Coarse-grained locks are easy to program but the parallelism is limited (DICE; SHALEV; SHAVIT, 2006). On the other hand, fine-grained locks provide good performance but they are hard to use (DICE; SHAVIT, 2007).

An alternative abstraction to replace mutual-exclusion locks in parallel programming is the *Transactional Memory* (TM) (HARRIS; LARUS; RAJWAR, 2010; GRAHN, 2010), in which critical-sections are accessed using transactions similar to the ones available in databases. With TM, instead of explicitly acquiring and releasing locks, the programmer only needs to delimit the block of code that he wants to be executed atomically as a transaction. The TM runtime is responsible to ensure a consistent execution, e.g., without deadlocks and race conditions. A transaction that has executed without conflicts can commit, i.e., update the memory with the new values. If a conflict was detected an abort is executed and a transaction is reinitialized until a commit is possible. Thus, an impression of atomicity is given to the programmer. Although there are TMs implemented in hardware (HTM) and in software (STM), this thesis focuses on the study of STM, where transaction consistency is guaranteed by a software library. An advantage of an implementation in software is that it is more flexible and not dependent on hardware. Also, it does not have the same resource limitations as in hardware (GRAHN, 2010).

1.1 Motivation

There are many compilers and programming languages that already support transactional memory constructs, such as C++ (since the standard C++11[1]), Haskell, Scala and .NET Framework (GRAMOLI; GUERRAOUI, 2017). Besides, some researches already showed that STM can outperform locks in some scenarios (DICE; SHAVIT, 2007; DRAGOJEVIĆ et al., 2011). Unfortunately, there are scenarios where the overhead added by the management of internal metadata of the STM or a high number of aborts, limits good performance (GRAMOLI; GUERRAOUI, 2014). Due to these limitations, improving performance of STM is an active research area. There are several proposals

[1] <https://gcc.gnu.org/wiki/TransactionalMemory>

for increasing the performance of STM systems. However, the majority of them focus on reducing the number of conflicts (transactional aborts). One technique is the use of a transactional scheduler, acting proactively, using heuristics to prevent conflicts and to decide *when* and *where* a transaction should be executed (DI SANZO, 2017).

Current multicore architectures have complex memory hierarchies and different latencies for memory accesses. Hence, a thread placement that improves the use of memory controllers and data locality is important to achieve good performance. A technique called *sharing-aware mapping*[2] (CRUZ; DIENER; NAVAUX, 2018) aims to map threads to cores and memory pages to NUMA nodes considering their memory access behavior. Since STM is used to synchronize data accessed by multiple threads, an efficient mapping will help to make better usage of caches and memory controllers, hence improving the overall performance. Besides, STM provides interesting mapping opportunities since the STM runtime has precise information about memory areas that are shared between threads, their respective memory addresses, and the intensity with which they are accessed by each thread. Hence, contrary to prior works on sharing-aware thread mapping, it is not necessary to keep track of all memory access of the applications, only the STM accesses. Therefore, the proposed mechanism will have a low overhead and can perform sharing-aware thread mapping accurately for STM applications.

1.2 Contributions

The main objective of this thesis is to investigate the use of sharing-aware mapping in the context of STM. Contrary to previous sharing-aware mapping proposals that rely on memory traces of the entire application, our proposal has lower overhead and better accuracy because only memory accesses that are in fact shared between threads are traced. Beyond that, this sharing-aware mapping will improve the overall performance of STM applications by improving the cache usage and interconnection traffic. More specifically, this thesis makes the following contributions:

- We developed a low overhead mechanism to **detect the sharing behavior of STM applications**, by tracking and analyzing how threads perform STM operations (Chapter 4).
- We made an **in-depth characterization of STM applications**, regarding memory access behavior, using the proposed mechanism. This characterization is used to define the suitability for a thread mapping based on communication behavior and defining which type of mapping policy is more appropriate (static or dynamic) (Chapter 5).

[2]This research field is also known as topology-aware mapping (JEANNOT et al., 2013; UNAT et al., 2017).

- We show how a **static thread mapping** (where threads are mapped to cores at the beginning of execution, and never migrated) is sufficient to improve the overall performance of the majority of STM applications (Chapter 6, Section 6.1).
- We extend the proposed mechanism to perform **online detecting sharing behavior and mapping**. We developed a heuristic to disable the mechanism if it determines that the application will not benefit from a new thread mapping (Chapter 6, Section 6.2).

1.3 Publications

The following papers were published during the PhD program and contain material that is relevant to this thesis:

1. Douglas P. Pasqualin, Matthias Diener, André R. Du Bois, Maurício L. Pilla. **"Thread Affinity in Software Transactional Memory."** 19th International Symposium on Parallel and Distributed Computing (ISPDC), July 2020 (PASQUALIN et al., 2020b).
2. Douglas P. Pasqualin, Matthias Diener, André R. Du Bois, Maurício L. Pilla. **"Online Sharing-Aware Thread Mapping in Software Transactional Memory."** 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), September 2020 (PASQUALIN et al., 2020a).
3. Douglas P. Pasqualin, Matthias Diener, André R. Du Bois, Maurício L. Pilla. **"Characterizing the Sharing Behavior of Applications using Software Transactional Memory."** Benchmarking, Measuring, and Optimizing (Bench'20). November 2020. (**Best Paper Award** and **Award for Excellence for Reproducible Research**) (PASQUALIN et al., 2021).

The following papers were also submitted for publication and are currently under peer-review:

1. Douglas P. Pasqualin, Matthias Diener, André R. Du Bois, Maurício L. Pilla. **"STMap: Sharing-Aware Thread Mapping in Software Transactional Memory."** Journal of Parallel and Distributed Computing (JPDC).
2. Douglas P. Pasqualin, Matthias Diener, André R. Du Bois, Maurício L. Pilla. **"Sharing-Aware Data Mapping in Software Transactional Memory."** SAMOS XXI - International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation.

1.4 Document organization

The remainder of this thesis is organized as follows. Chapter 2 presents the background of different topics used in this thesis, such as TM and sharing-aware mapping. Chapter 3 presents the works related to the thesis subject. Also, we discuss the differences between our contributions with related work. Chapter 4 presents the first contribution of this thesis, i.e, a mechanism to detected the sharing behavior of STM applications, by tracking and analyzing how threads perform STM operations. Chapter 5 uses the proposed mechanism of Chapter 4 to perform an in-depth characterization of `STAMP`, a frequently used TM benchmark suite, regarding memory access behavior. It includes information about the suitability for thread mapping of each `STAMP` application, its communication pattern, and its dynamic behavior, among others. Chapter 6 shows how to use the proposed mechanism to perform static and online thread mapping based on the memory access behavior of STM operations. Finally, Chapter 7 presents the conclusion and future work. We also included Appendix A which shows how to extend the online sharing-aware thread mapping mechanism to include data mapping.

2 BACKGROUND

This chapter covers the background on the range of different topics which are important to this thesis. It starts by explaining transactional memory (Section 2.1) and sharing-aware mapping (Section 2.3). Benchmarks for STM are briefly described in Section 2.2. We also include a small experiment to show the benefits of sharing-aware thread mapping for STM applications (Section 2.4).

2.1 Transactional Memory

Transactional memory (TM) is an abstraction to synchronize accesses to shared variables. Instead of using locks, the programmer only needs to enclose the critical section in an atomic block, which will be executed as a transaction. The concept of transactions was borrowed from Databases. In fact, the first idea to use Database transactions in programming languages was described by Lomet (1977). Sixteen years later, Herlihy and Moss (1993) proposed hardware support for TM. The first implementation purely on software (STM) was proposed by Shavit and Touitou (1995) as a flexible alternative not dependent on hardware. There are also hybrid approaches (DAMRON et al., 2006) that combine implementation both on hardware and software.

2.1.1 General Concepts

The execution of a transaction needs to be *atomic*. Atomicity requires that a transaction is executed as a whole or it needs to appear as it was never executed (HARRIS; LARUS; RAJWAR, 2010; GRAHN, 2010). This property is also known as “all-or-nothing” (ÖZSU; VALDURIEZ, 1996). A transaction *commits* if executed without conflicts, hence all operations and results are made visible to the rest of the system (GRAHN, 2010). If conflicts are detected, a transaction *aborts*, i.e., all operations are discarded, and the transaction needs to restart until a commit is possible. This idea is associated with another important property called *isolation*: all memory updates of a running transaction can not be visible to other transactions before a commit.

The sequence of operations performed by all transactions in a given execution is

called *history* (GUERRAOUI; KAPALKA, 2008). If one transaction executes only after the end of the other they are serial, otherwise *concurrent* (HARRIS; LARUS; RAJWAR, 2010). If they are concurrent, conflicts could occur between them. A conflict occurs when two transactions perform operations in the same memory location, and at least one of these operations is a write.

As transactions can execute concurrently, correctness criteria were proposed to ensure that the TM system produces correct results. One of the most used criteria is *Opacity* (GUERRAOUI; KAPALKA, 2008), that is an extension of the classical database criteria *Strict Serializability* (PAPADIMITRIOU, 1979). Serializability says that the result of executing concurrent transactions in a given history must be equal to a serial execution. *Strict Serializability* says that real-time order must be respected. If T_1 finishes before T_2 starts, then T_1 must occur before T_2 in the equivalent serial execution (HARRIS; LARUS; RAJWAR, 2010). Opacity extends these concepts to aborted transactions, i.e., all transactions in a given history, including aborted, must appear to be executed in a serial order.

2.1.2 Design Choices

Although the main purpose of TM is to provide a simple interface to manage accesses to shared resources, its implementation is not trivial. Many different design options are available such as transaction granularity, version management, conflict detection and resolution. The next subsections describe these design options.

2.1.2.1 Version Management

TMs use version management of memory locations to manage the writes of concurrent transactions. Two approaches are used (HARRIS; LARUS; RAJWAR, 2010; FELBER et al., 2010):

- **Eager**, **direct update** or **write-through**: data is modified directly in memory. Older values are stored in an *undo-log*. In case of an abort the log is used to restore the old values.
- **Lazy**, **deferred update** or **write-back**: instead of updating data directly in memory, new values are stored in an *redo-log*. During a commit, the log is used to set new values to memory. In case of an abort, the log is discarded.

Eager versioning makes committing faster, whereas lazy versioning makes aborting faster (GRAHN, 2010).

2.1.2.2 Transaction Granularity

The granularity is the dimension used for conflict detection, i.e., the level used for keeping track of memory locations. One option is to use memory **word** granularity. The

main advantage is that no false conflicts happen. However it adds a high overhead in terms of time and space to keep the metadata (GRAHN, 2010). **Object** granularity is most used in object-based languages (CASTRO, 2012). However it can lead to false conflicts, i.e., transactions accessing the same object but different fields (LARUS; KOZYRAKIS, 2008). For HTM, **cache line** granularity is the most suitable (GRAHN, 2010).

2.1.2.3 Conflict Detection

Similar to version management, there are two approaches to deal with conflict detection between concurrent transactions (GRAHN, 2010):

- **Eager**, **early**, **pessimistic** or **encounter-time**: conflicts are verified on each memory location read or written exactly when it occurs. In STM this could be done using locks or version number on memory locations. Thus, to access a value, a transaction needs to acquire its ownership, preventing others from accessing it (BANDEIRA et al., 2015).
- **Lazy**, **late**, **optimistic** or **commit-time**: conflicts are verified only at commit time. It allows multiple transactions to access shared data and continue executing even if they conflict, as the TM system will detect and resolve them on commit time (HARRIS; LARUS; RAJWAR, 2010, p. 20).

These options could be combined, for instance, lazy conflict detection for reads and eager for writes (BANDEIRA et al., 2015).

2.1.2.4 Conflict Resolution

All operations executed by a transaction that aborts could be seen as a wasted work (SPEAR et al., 2009; ANSARI et al., 2009; ZHOU et al., 2016). Thus, the total aborts in an execution have a strong relationship with the final performance. In case of conflicts, choosing which transaction needs to be aborted is a responsibility of the *Contention Manager* (CM) (YOO; LEE, 2008). When two transactions T_A and T_B conflict, the transaction that has detected the conflict, suppose T_A, asks the CM what to do. The actions could be, for instance, abort immediately or wait a determined time, allowing T_B to finish, or force and abort of T_B (GUERRAOUI; HERLIHY; POCHON, 2006). There are many CMs proposed in the literature with different purposes. A few examples of them are (SCHERER III; SCOTT, 2005; SPEAR et al., 2009; HARRIS; LARUS; RAJWAR, 2010; GRAHN, 2010):

- **Passive**: the transaction that detected the conflict aborts and restart its execution.
- **Polite**: the transaction that detected the conflict could abort the conflicting one. However, aborting is delayed for a period of time, waiting the conflicting transaction

to end accessing values. The transaction can also wait for a fixed number of exponentially growing intervals before aborting the enemy.

- **Karma**: use priorities to define which transaction must abort. The priority is defined by the total of memory locations that a transaction has accessed. The total is cumulative, i.e., taking in consideration all aborts and re-executions.
- **Timestamp**: aborts the transaction that started earlier.
- **Polka**: a hybrid between *Polite* and *Karma*.

2.1.3 STM Implementation

The first STM implementations were non-blocking (SHAVIT; TOUITOU, 1997), more specifically obstruction-free (HERLIHY et al., 2003). However, according to Ennals (2006), obstruction-free is essential in distributed systems but not appropriate for non-distributed STM. In the same publication, Ennals shows that lock-based STM are simpler to implement and faster. As a consequence many state-of-art STM implementations, for instance, `TL2` (DICE; SHALEV; SHAVIT, 2006), `TinySTM` (FELBER et al., 2010) and `SwissSTM` (DRAGOJEVIĆ; GUERRAOUI; KAPALKA, 2009) are lock-based.

Other characteristic used in the first implementations was *visible reads*, i.e., all transactions knew who read a specific memory location. To implement this approach it is necessary a list to store all transactions who read a specific memory location. Thus, when a transaction writes new values in memory, it is possible to notify the readers that there is a conflict. The disadvantage of this approach arises when many distinct threads read the same memory location (FELBER et al., 2010). As most workloads are read-intensive, this approach limits the performance (SUTRA et al., 2018). The opposite solution is to use *invisible reads*, where other threads do not know who read a specific memory location. However, using this approach, a thread could be in an inconsistent state (due to a conflict) and does not know yet. Letting this transaction continue execution could bring an unwanted result, not guaranteeing Opacity or other consistency. Thus, a solution to using invisible reads is to periodically validate the read set, verifying if the memory locations read are still consistent. On the other hand, this solution is expensive, mainly if a transaction has read many memory locations. This cost keeps growing as the transaction keeps reading different memory locations. This problem is known as *incremental validation* (HARRIS; LARUS; RAJWAR, 2010).

To use invisible reads without the problem of incremental validation the concept of *global clock* was introduced in an independent way in the algorithms `TL2` (DICE; SHALEV; SHAVIT, 2006) and `LSA` (Lazy Snapshot Algorithm) (RIEGEL; FELBER; FETZER, 2006). The global clock is a counter utilized for versioning memory locations. When a transaction writes new values to memory, the global clock is incremented and

the new clock value is used as a version number for modified memory locations. In STM implementations that use a global clock, transactions still need to validate their read set, but in general, this is necessary only for write transactions during the commit phase. Read only transactions can commit without validation, because memory location were validated when read.

2.1.3.1 Using a global clock

This section presents an overview of how an STM implementation that uses a global clock for versioning memory locations works. The operations described are based on algorithms `TL2` (DICE; SHALEV; SHAVIT, 2006) and `TinySTM` (FELBER et al., 2010). However, minor details could be different depending on the STM implementation.

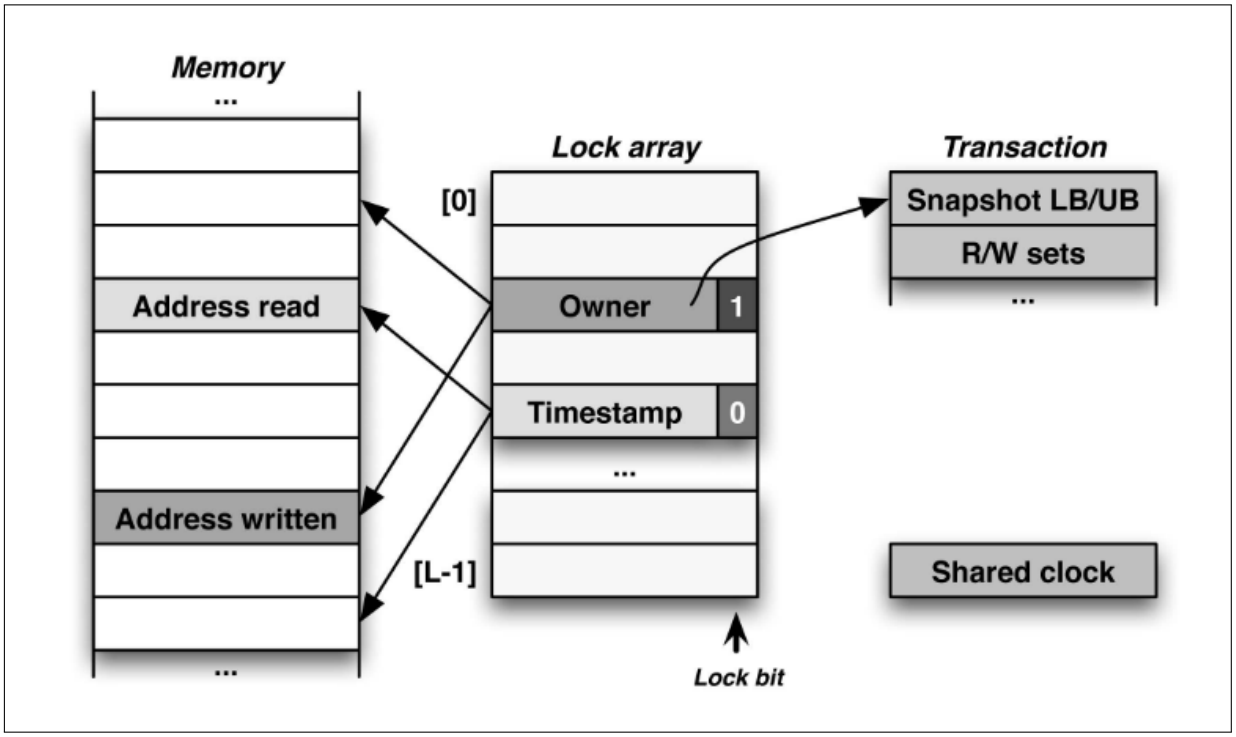


Figure 1 – Data structures utilized internally by the STM library `TinySTM`. Source: (FELBER et al., 2010). R/W sets stands for read and write sets. A snapshot corresponds to a range of valid linearization points. LB and UB stand for lower and upper bounds, i.e, the validity range of the snapshot.

Figure 1, represents the internal organization of `TinySTM`. Each transaction has two internal linked lists, called read-set (RS) and write-set (WS). These lists are utilized to keep track of memory locations read and written by transactions. Another important data structure for STMs, is the lock array, implemented using a hash table. This table is utilized to map memory locations to their related versioned locks. When a versioned lock is locked, its last bit is set to one and the remaining bits contain the owner of the memory location. If the lock is free, it contains the current version of the memory location, also called timestamp (Figure 1). Some authors call the lock array as *owner-*

ship records (orecs), because it associates memory locations with their current owners (DALESSANDRO; SPEAR; SCOTT, 2010).

To summarize, a transaction performs the follow operations. To make it simpler, we do not intent to cover all possible algorithm cases. The main idea is to show the basics of how an STM algorithm works.

- **Begin Transaction**: The current global clock value is copied to a local variable of the transaction. Normally, the variable name is rv which stands for ***r**ead **v**ersion*. This value will be utilized to validate reads on memory locations.

- **Read** or **Load**: With **eager** version management (Section 2.1.2.1), the transaction verifies if it owns the lock of the memory location that is being read. If true, it returns directly the value stored in memory. If the lock of the memory location belongs to another transaction, an abort is necessary. Using **lazy** version management, the transaction first verifies if the memory location that it wants to read is in their WS. In that case, the transaction has already written to this location and the value in the WS must be returned. Otherwise, it is verified if the memory location is locked by another transaction. If true, an abort is necessary. Otherwise, the version associated with the memory location is compared with rv. If the version of the memory locations is greater than rv, an abort is necessary, as the memory location was updated after the transaction that is accessing it started. In case of abort, independently of the version management, the CM (Section 2.1.2.4) is triggered. STM implementations, such as `TinySTM`, try to do additional processing before aborting a transaction, for instance, the use of *timestamp extension* technique (RIEGEL; FELBER; FETZER, 2006). Finally, the address read is added to the RS and the content of the memory location is returned.

- **Write** or **Store**: First, it is necessary to verify if the memory location is locked. If true an abort is necessary. The next step, in case of **eager** conflict detection (Section 2.1.2.3) is to lock the memory location. If **eager** version management (Section 2.1.2.1) is utilized, the new value is updated directly on memory and the old one is stored in an undo-log. If the version management is **lazy**, the new value is stored in the WS to be updated at commit time.

- **Commit**: Read-only transactions can commit directly, as the memory locations were validated on the **Read** step. For write transactions, the first step is to validate the RS, verifying each address, if it is locked by other transaction and if the rv is still valid. If the implementation uses **lazy** version management (Section 2.1.2.1), all address in the WS should be locked. If it fails, the transaction aborts. In case of eager version management, addresses have already been locked in the **Write** step. Then, the global clock is advanced by 1 and the result is used as

Table 1 – Applications from STAMP benchmark. Source: (MINH et al., 2008).

Application	Domain	Description
bayes	machine learning	Learns structure of a Bayesian network
genome	bioinformatics	Performs gene sequencing
intruder	security	Detects network intrusions
kmeans	data mining	Implements K-means clustering
labyrinth	engineering	Routes paths in a maze
ssca2	scientific	Creates efficient graph representation
vacation	online transaction processing	Emulates travel reservation system
yada	scientific	Refines a Delaunay mesh

the new version for the values being updated. If **lazy** version management is used, the new values are updated in the memory. In the case of **eager** version management, the new values were updated in the **Write** step. A final step, for both version managements, is to release the locks and set the new version value of the memory location.

- **Abort**: If the implementation uses eager version management, all values from the *undo-log* must be restored. Otherwise, the *redo-log* is discarded. A final step is to release all locks, if acquired.

2.2 Benchmarks for STM

Together with the first STM proposals, researches also developed benchmarks for testing. In most of the cases, these benchmarks were simple, based on sets, lists and maps (HERLIHY et al., 2003; HARRIS; FRASER, 2003; SCHERER III; SCOTT, 2005; RIEGEL; FELBER; FETZER, 2006; DICE; SHAVIT, 2007). Hence, it was necessary to develop specific TM benchmarks to evaluate TM systems. More specifically, benchmarks with realistic characteristics. One of the first benchmarks proposed for evaluating STM systems were STMBench7 (GUERRAOUI; KAPALKA; VITEK, 2007) and Lee-TM (ANSARI et al., 2008c). After that, other suites and standalone benchmarks applications were proposed to evaluate TM systems, for instance, Eigenbench (HONG et al., 2010), RMS-TM (KESTOR et al., 2011) and, Memcached (RUAN et al., 2014).

Despite the effort on STM benchmarks proposal, the most used for evaluating TM implementations is the STAMP (*Stanford Transactional Applications for Multi-Processing*) (MINH et al., 2008). This suite is composed of 8 applications with realistic characteristics and that represent several application domains. Table 1 shows the domain and a short description of each application from STAMP benchmark.

STAMP still is the most used STM benchmark suite, as can be seen in recent researches (CHEN; GIBBONS; MOWRY, 2020; CARVALHO et al., 2020; DI SANZO et al., 2020; YU; ZUO; ZHAO, 2019; POUDEL; SHARMA, 2019; MURURU; GAVRILOVSKA;

PANDE, 2019).

2.3 Sharing-Aware Mapping

Data locality is an important factor in modern multicore and NUMA systems. One way to better explore locality is to map threads and data according to their *memory access behavior* (DIENER et al., 2016a). Hence, two types of mappings are possible (CRUZ; DIENER; NAVAUX, 2018):

1. **Thread mapping**: threads are associated to cores, improving the cache usage and interconnections, i.e., threads are mapped to cores that are close to each other in the underlying architecture.

2. **Data mapping**: memory pages are associated with NUMA nodes, optimizing the usage of memory controllers, i.e., memory pages are mapped to the same NUMA node where the core that is accessing them belongs.

Thread and data mapping based on the memory access behavior of applications is called **sharing-aware mapping** (CRUZ; DIENER; NAVAUX, 2018). Although the Linux kernel handles thread and data mapping, for thread mapping it does not take memory access patterns into consideration. For instance, the *Completely Fair Scheduler* (CFS) (WONG et al., 2008) used by default in the Linux kernel (DIENER et al., 2016a) mainly focuses on load balancing. For data mapping, the default policy is called *first-touch* (GAUD et al., 2015) where the memory is allocated in the NUMA node where the first access to the memory page is performed. Another data mapping policy available is *interleave*, that focuses on balance, allocating pages in a round-robin way on the NUMA nodes (LAMETER, 2013).

To perform a thread mapping, it is necessary to know how threads share data (DIENER et al., 2016a). This information is usually represented as a *communication matrix* (BORDAGE; JEANNOT, 2018). Also required is information about the hardware hierarchy, which can be discovered using tools such as `hwloc` (BROQUEDIS et al., 2010a). A mapping algorithm uses the communication matrix and hardware hierarchy to choose an improved mapping of threads to cores.

For data mapping based on memory access, the accesses of each NUMA node to pages must be known (CRUZ; DIENER; NAVAUX, 2018). Current systems have millions or even billions of memory pages. For this reason, only a small group of pages must be considered for the mapping decision, avoiding a high overhead (DIENER et al., 2016a).

As mentioned in Section 1.1, STM provides interesting mapping opportunities since the STM runtime has precise information about memory areas that are shared between threads. The main idea of this thesis is to use information about transactional shared

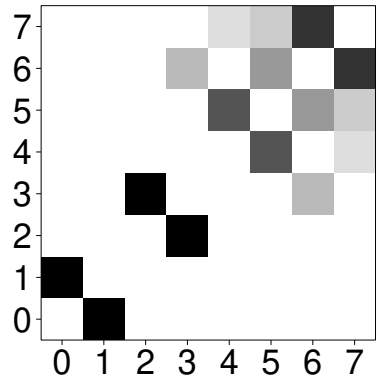
variables inside STM runtime to perform an efficient mapping. Hence, this thesis will focus on sharing-aware **thread mapping**. For efficient data mapping is necessary to have a global vision of the memory pages of an application, not only the ones accessed by the STM runtime. Nevertheless, in Appendix A we made some experiments with sharing-aware data mapping in STM to confirm this hypothesis.

2.3.1 Communication/sharing matrix

To determine a better placement of threads and data, an affinity measure is required. For thread mapping, a common measure is a communication or sharing matrix (BORDAGE; JEANNOT, 2018; MAZAHERI; WOLF; JANNESARI, 2018), in which each cell represents the amount of communication between pairs of threads (SASONGKO et al., 2019). Since the amount of communication between thread i and j is the same between j and i, the communication matrix is symmetric and diagonals are zero (MAZAHERI; WOLF; JANNESARI, 2018). Figure 2 shown examples of communication matrices, where axes show thread IDs. In Figure 2(b), the matrix is represented graphically,

	0	1	2	3	4	5	6	7
7	0	0	0	0	1	2	10	0
6	0	0	0	3	0	5	0	10
5	0	0	0	0	8	0	5	2
4	0	0	0	0	0	8	0	1
3	0	0	12	0	0	0	3	0
2	0	0	0	12	0	0	0	0
1	12	0	0	0	0	0	0	0
0	0	12	0	0	0	0	0	0

(a) Numbered matrix.



(b) Colored matrix (heatmap).

Figure 2 – Examples of communication matrices

where darker cells indicate more communication between pairs of threads (DIENER et al., 2016b).

2.4 Improving STM applications with Thread Mapping

To show how sharing-aware thread mapping can improve the performance of an STM application, we designed an experiment that illustrates sharing-aware mapping of an STM application that calculates the sum of 16 million array elements. In the application, each group of 2 threads computes the sum of their respective array part in a shared sum variable. For example, with 8 threads, there are 4 shared variables for computing the sum. The memory access behavior is known in advance: threads 0 and 1 access a shared variable, threads 2 and 3 another shared variable, and so on. Algorithm 1 shows how the sum function works. Lines 5-12 verify the number of

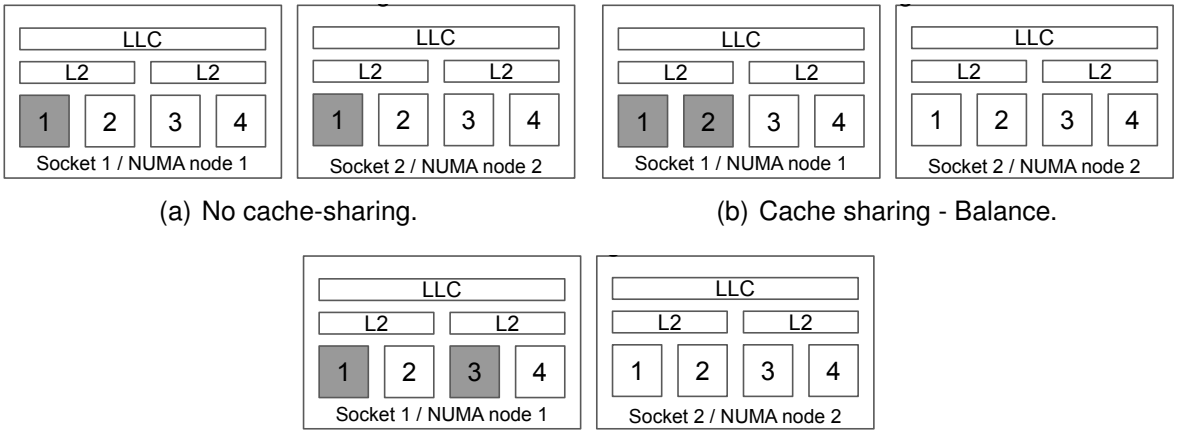
Algorithm 1 Function executed by each thread on the array sum application

```
1: function SUM
Require:
    tid: thread ID that is accessing this function

2:      pinThreadToCore(tid, coreId)          ▷ bind thread to core according to the strategy
3:      for all value in array do
4:          stm_start_transaction()                                   ▷ begin transaction
5:          if (tid in 0, 1) then
6:              stm_write(sum_0_1, stm_read(sum_0_1) + stm_read(value))
7:          else if (tid in 2, 3) then
8:              stm_write(sum_2_3, stm_read(sum_2_3) + stm_read(value))
9:          else if (tid in 4, 5) then
10:             stm_write(sum_4_5, stm_read(sum_4_5) + stm_read(value))
11:         else if (tid in 6, 7) then
12:             stm_write(sum_6_7, stm_read(sum_6_7) + stm_read(value))
13:         stm_commit()                                              ▷ try to commit
```

the thread that is performing the sum and stores it in the respective shared variable. Thus, keeping threads that share a sum variable on sibling cores improves the cache usage. We used the `TinySTM` (FELBER et al., 2010) library for the synchronization of shared variables, with the default configuration: *lazy* version management, *eager* conflict detection and CM *suicide*.

We executed this application on the following NUMA machines: *Xeon*, with 8 Intel E5-4650 processors, totaling 96 cores and 8 NUMA nodes and *Opteron*, with 4 AMD Opteron 6276 processors, totaling 64 threads and also 8 NUMA nodes (more details on the machines in Section 6.1.1). For the tests, 4 different configurations were used: the default “Linux scheduler”, “no cache sharing”, “Cache sharing - Balance” and “Cache sharing - Socket”. With exception of “Linux scheduler”, the mapping strategies are shown in Figure 3.



(a) No cache-sharing.

(b) Cache sharing - Balance.

(c) Cache sharing - Socket.

Figure 3 – Thread mapping strategies for the Array Sum application.

The idea of “no cache sharing” is to map threads that share the same variable to different NUMA nodes, forcing remote accesses and cache coherency messages between the nodes. In contrast, in the “Cache sharing - Balance” approach, the idea is to map threads that share a variable to sibling cores on the same NUMA node to share caches. However, for this configuration we map each pair of shared variables to different sockets.

Finally, for “Cache sharing - Socket”, the idea is to place threads on sibling cores, sharing all cache levels. Since this application uses 8 threads, using this configuration all threads will be mapped to only one socket. To pin threads to cores the function `pthread_setaffinity_np` was utilized. The mapping was applied when a thread calls the function `stm_init_thread` of the `TinySTM` library. This function informs the STM runtime that the thread that has called it will perform transactional operations. Figure 4 presents the execution time in seconds on each machine.

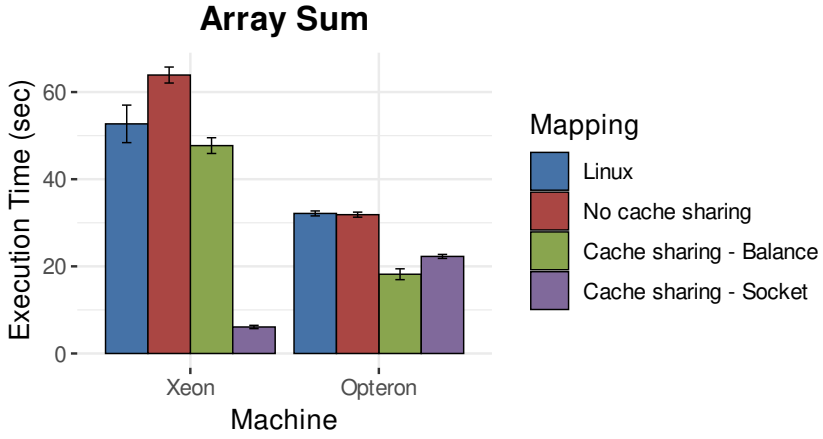


Figure 4 – Execution time of the Array Sum application.

On the Opteron machine, the Linux scheduler had similar results as the “no cache sharing” configuration. When the application was running, we observed that the scheduler tries to balance threads, distributing them to the NUMA nodes, without taking the sharing behavior into account. This explains that the results are similar to “no cache sharing”. On Xeon, the NUMA effects are more clear since forcing threads to run on distinct sockets, i.e., “no cache sharing” configuration, presented the worst performance.

For the “Cache sharing - Balance” configuration, on Xeon the execution time was reduced by 9.46% whereas in Opteron it was reduced by 43.02%. However, the most interesting result appears using “Cache sharing - Socket” mapping on the Xeon. Using this configuration, the execution time was reduced by 88.46%. Although in Opteron the result of “Socket” was also positive (30.69% of reduction time), the “Balance” configuration was better. We think that the good results of “Socket” on Xeon can be explained by the size of the last-level cache (LLC). The size of the data structures used by the array sum is 64MB. The size of LLC of the Xeon is 30MB whereas in Opteron it is

6MB. In that case, almost half of the memory used by the application fits entirely on the LLC of the Xeon. For the Opteron, mapping the variables to distinct cores increases the performance gains compared to “Socket” configuration, because more cache was available to the application.

2.5 Summary

This chapter presented the background related to this thesis. It also included a small experiment with a synthetic application to illustrate the possible benefits of sharing-aware thread mapping for STM applications.

3 RELATED WORK

This chapter presents the works related to the thesis subject. Section 3.1 describes transactional schedulers, one technique used to reduce the number of aborts during transactional execution, hence, improving the performance. Recent works use scheduling techniques for STM with the objective of optimizing resources, like cache and page sharing or to reduce the latency on data access. This kind of schedulers are described in Section (3.2), which presents an exhaustive list of works that uses thread and data mapping to improve the performance of STM applications. The next Section (3.3) also describes works that explore thread and data mapping to improve the performance of applications that do not use STM. To perform a successful mapping it is necessary to perform an in-depth analysis of STM applications, regarding sharing behavior. Hence, Section 3.4 describes works that perform workload characterization of STM applications and sharing behavior of general applications.

3.1 Transactional schedulers

Albeit a Contention Manager (CM) can help to reduce the number of aborts in a transactional execution, it has limitations. The CM acts only in a **reactive** way, dealing with a conflict when it occurs and not avoiding it (YOO; LEE, 2008; DRAGOJEVIĆ et al., 2009; NICÁCIO; BALDASSIN; ARAÚJO, 2012). Transactional scheduling acts in a **proactive** way, using heuristics to prevent conflicts and to decide *when* and *where* a transaction should be executed (DRAGOJEVIĆ et al., 2009).

This section presents state-of-the-art transactional scheduling techniques. In order to identify the works on this subject, the surveys (HENDLER; SUISSA-PELEG, 2015) and (DI SANZO, 2017) were used as a basis. Also, this section describe works published after the surveys. However, transactional schedulers focused on HTM, GPU (Graphics Processing Unit) or real-time systems are not included, because they are not on the scope of this thesis. Di Sanzo (2017) classifies the schedulers as reactive, prediction-driven, feedback-driven (all heuristic based) and machine learning or analytical (based on performance models (TAY, 2018)). In the next sections, we will follow this

classification to describe the transactional schedulers.

3.1.1 Feedback-driven

Feedback techniques are based on constantly comparing if the *actual behavior* of a system is the *desired* (JANERT, 2013). Application parameters are monitored during the execution of a transaction (*actual behavior*), and in each new iteration, they are used as input for the scheduler to take decisions, i.e., a corrective action in order to achieve the *desired behavior*. This step is repeated during execution, trying to dynamically adjust the application (DI SANZO, 2017).

One of the first works to propose a transaction scheduler for STM was the *Adaptive transaction scheduling* (ATS) (YOO; LEE, 2008), and the idea was to work together with the CM: each thread has a *contention intensity* (CI), recalculated each time that a transaction finishes (commit or abort). When the CI is great than a predefined threshold, threads are inserted in a global queue to be serialized. This approach takes into consideration that when the CI is high, it is better to limit parallelism, avoiding possible conflicts.

In Ansari et al. (2008a), an adaptive concurrency control (ACC) technique was proposed which limits the maximum number of concurrent threads executing transactions. The idea is that an excessive number of threads can hurt the performance in a high contention environment, mainly due to a higher number of aborts. The technique keeps track of a *Transaction Commit Ratio* (TCR), i.e., the percentage of committed transactions in the total number of transactions executed, and uses it to dynamically adapt the number of concurrent threads. The ACC uses two parameters: a *target TCR range* and a time *interval* for calculating the TCR. Four adaptive concurrency control algorithms were proposed, varying the heuristic to change the total number of active threads, but all based on TCR.

In Ansari et al. (2008b), the authors proposed a new concurrency control algorithm, called *P-only Concurrency Controller* (PoCC), extending the ACC work. The main idea is to keep TCR at a configurable value (called *set point*) instead of a range as in the previous work. If a high *set point* is chosen, then the number of threads will quickly reduce when TCR decreases. On the other hand, it will have a slower response when TCR grows suddenly.

Chan, Lam and Wang (2011) also propose a new concurrency control technique. For this purpose, a parameter *quota* is recalculated every predetermined amount of time. When a new transaction is to be started, it needs to check if there is sufficient quota available, otherwise, it will wait. There are two proposals for calculating the *quota*. The first one, called *Throttle*, adjusts the quota based on the commit *ratio*, which is compared with a predefined threshold, and the quota is adjusted according to it. The second one, called *Probe*, uses commit *rate* (total of commits per unit of time) instead

of *ratio*. Also, it uses a try and error approach to find the best quota, instead of a fixed threshold.

Ansari (2014) published another work using TCR as the main heuristic. The difference from their first work (ANSARI et al., 2008a) and PoCC is that in this new algorithm, called *Weighted Adaptive Concurrency Control* (WACC), the TCR is calculated per thread. Similar to PoCC, that tried to keep TCR at a configurable value (*set point*), WACC uses the notion of *expected TCR*. It predicts the global TCR that should be achieved if the determined subset of threads is activated. Thus, it is possible to predict if the *set point* will be reached if the determined subset of threads will be allowed to concurrently run. Like in the first work (ANSARI et al., 2008a), four approaches are proposed, using different heuristics (all based on TCR) to define the total number of threads.

One of the main objectives of Rito and Cachopo (2014), is to avoid excessive serialization using a fine-grained approach. They have proposed the ProPS (*Progressively Pessimistic Scheduling*) technique. In ProPS, a global *concurrency level* (CL) matrix keeps information about pairs of atomic operations i and j. For the authors, each transaction that executes the same block of code, execute the same operation of type i. Accessing the matrix on the index CL_{ij}, it is possible to know how many transactions executing atomic operations of type i could be executed concurrently with another one executing j atomic operations. In the beginning, the values in the matrix are set to be equal to the maximum of threads or cores of the machine, i.e., all cores could execute all types of transactions without restriction. When a specific CL_{ij} decreases (typically by an abort of i or j), ProPS reduces the number of transactions executing i and j. By design, ProPS reduces exponentially (*progressively pessimistic*) the concurrency on aborts between atomic operations and increases it linearly at commit.

The idea proposed by Pereira, Amaral and Araújo (2014) is to use the *percentage of effective work* (PEW) of a transaction as the main heuristic. The PEW replaces CI in approaches like ATS. It is calculated for each transaction, and it is based on the total of cycles executed until the transaction finishes (commit or abort). Transactions with less effective work done are prioritized. If a transaction aborts and the PEW is high, then it is wasting too many work (cycles) and should have a lower priority. There are queues of transactions according to their priorities. The authors noted that only using PEW, there is still a high number of conflicts between transactions with the same priority. Thus, an additional heuristic was included: a *success-reward policy* (SRP). According to a predefined *reward* threshold, a transaction could change its position in the queue.

In Ravichandran and Pande (2014), the authors classified applications as *fully scalable* and *scalability limited*. Applications under the former classification decrease their execution time as the number of cores of the machine grows. The latter is the opposite, hurting the application performance as the number of cores is increased. The authors have proposed F2C2-STM (Flux-based Feedback-driven Concurrency Control), which

focuses on *scalability limited* applications. F2C2-STM is inspired by TCP's (Transmission Control Protocol) network congestion control algorithm, to adjust the maximum number of concurrent threads allowed to run. The heuristic utilized is the transaction throughput. However, to compute the global throughput between all threads, a global variable is needed. To avoid synchronization, the authors chose to calculate the throughput only in one thread. Thus, they have assumed that the transactional workload is roughly the same between all threads. To find the best concurrency level, like in Chan, Lam and Wang (2011), they have used a try and error approach, monitoring the transaction throughput on each modification.

3.1.2 Reactive

Reactive techniques are activated after a conflict is detected. The main objective is to avoid the same conflict to occur again (DI SANZO, 2017).

The *Collision Avoidance and Resolution* (CAR-STM) scheduling-based mechanism (DOLEV; HENDLER; SUISSA, 2008) presents two new CMs and a scheduler. Each core has one queue, and the runtime system restricts the maximum number of threads to be equals to the cores available. The first CM proposed is called *Basic*. When it detects a conflict between two transactions, it aborts the newer transaction and move it to the queue of the older. The second, called *Permanent*, also aborts the newer transaction in case of conflicts. However, it marks the aborted as a subordinate of the older. If a transaction needs to be moved to another queue, *Permanent* will also move all its subordinates. A proactive centralized module tries to choose the queue for a new transaction, based on a conflict probability with the running transactions on each queue.

On Steal-on-Abort (SoA) (ANSARI et al., 2009), the idea is to find dynamically an optimal order to execute transactions, minimizing aborts. Each thread has two queues. The first one, called main queue, keeps new transactions that should be processed by the thread. The second queue is called "steal". When a transaction aborts due to a conflict, instead of restarting immediately, the opponent transaction "steals" the aborted one and put in the steal queue of the thread. When the opponent transaction commits, transactions on the steal queue are moved to the main queue. The authors proposed different strategies to choose in what position of the queue the stolen transaction will be put in. According to the authors, SoA benefits applications that have a high number of repeated conflicts.

In Attiya and Milani (2009, 2012), the focus of the proposed schedulers are workloads with read-dominated transactions and lazy versioning. The authors explain that many proposed schedulers focuses on avoiding repeated conflicts, serializing transactions. Also, they do not perform well under read-dominated workloads, serializing more transactions than necessary. With this motivation, the BIMODAL scheduler is proposed.

On BIMODAL, each core has a work queue. Also, a global *RO-queue*, for read-only transactions is shared between all cores. When two transactions conflict, if the aborted is a writing one, it will be moved to the same queue of the conflicting. If it is read-only, it will be moved to the RO-queue. When the RO-queue reaches a threshold of enqueued transactions (or the other queues are empty), the BIMODAL will prioritize the transactions in the RO-queue. The scheduler is proposed in a theoretical way. It was formally proved and compared with SoA.

Sharp and Morgan (2013) argue that prior transactional schedulers deal with concurrent conflicts only, i.e., between a reader and a writer. The objective of the authors is to deal with *semantic conflicts*. This kind of conflict occurs when there is no concurrent conflict and yet it is not possible for transactions to proceed. They have cited as an example a transaction that needs to consume an item from a buffer but the buffer is empty, or a withdraw in an account without funds. With this motivation the Hugh scheduler is proposed. When a transaction aborts, it first needs to register itself in a *transaction table*. Next, a speculative phase begins, executing a permutation of transactions that is in the *transaction table*. This phase tries to find a permutation where the maximum of pending transactions can commit. Finally, in the commit phase, all successful permutations are sent to an algorithm to decide which permutation can commit. It chooses the permutation with the greatest number of transactions. The authors also proposed Hugh2 (SHARP; BLEWITT; MORGAN, 2014), using the same idea on a different STM implementation.

The *second-hop conflict* is a concept proposed in the RelSTM scheduler (SAINZ; ATTIYA, 2013). If a transaction T_{1h} conflicted with T_x, and another T_{2h} has conflicted with T_{1h}, then T_{2h} is a *second-hop* of T_{1h}. Upon a conflict, the transaction registers its opponent and those that have conflicted with the opponent too. When restarting, the transaction is serialized if the opponents are still running. Otherwise it could wait if a percentage of *second-hop* transactions that are still running is greater than a predetermined threshold.

3.1.3 Prediction-driven

Schedulers under this classification, use prediction techniques, trying to increase the probability to make the right decision on scheduling (DI SANZO, 2017).

The Shrink scheduler (DRAGOJEVIĆ et al., 2009) tries to predict the future memory accesses of transactions based on past accesses. The authors use the concept of temporal locality of the read-set. They have identified that multiple consecutive committed transactions in a thread access similar addresses. A per thread *Bloom Filter*[1] keeps track of the last read addresses. When an address is read, Shrink verifies if it is in the

[1] *Bloom Filter* is a probabilistic data structure which allows fast search and insertion of data (BLOOM, 1970).

Bloom Filter. If true, it was recently accessed and it is included in the *predicted read-set* of the thread. In case of abort, the write-set is copied to the *predicted write-set*. Each thread has a *commit rate*, calculated every time that a transaction commits or aborts. When the commit rate is greater than a predefined threshold, `Shrink` verifies, before starting a transaction, if some address in the prediction sets (read or write) are being written by other threads. If true, the serialization is activated.

In Heber, Hendler and Suissa (2009, 2012), the focus is to avoid too much oscillation between serialization and non-serialization periods. The authors proposed a mechanism called *Low-Overhead Serializing* algorithm (LO-SER) which aims to keep certain stability between serialization times. The scheduler collects statistics like past commits and aborts. This data can be local (per thread) or global. The proposed stabilizing mechanism calculates a *contention level* (CL) each time that a transaction aborts or commits. Then, a low (lt) and a high threshold (ht) needs to be defined, i.e., a range. Serialization is applied if CL is greater than ht and deactivated when CL is lower than lt.

Atoofian (2011) uses the intuition that if a transaction aborted, it will fail again in the future. This theory is called *locality of contention*. To avoid it, it is important that the transaction that caused the abort should finish before restarting the aborted. With this motivation, the *Speculative Contention Avoidance* (`SCA`) mechanism is proposed. Each thread has a Contention Predictor (CP), composed by a *contention bit* (CB) and a *saturating counter* (SC). The CB shows the result of the last transaction executed on the thread (1 if failed or 0 if committed). The SC is similar to branch predictors used in pipelines processors (YEH; PATT, 1992). It is incremented each time that a transaction conflicts and zeroed when commits. Before a thread executes, it consults the `SCA`, that will serialize the transaction if CB is equals to 1 and SC is greater than a given threshold.

3.1.4 Mixed Heuristics

This section describes works that use mixed strategies according to different transaction profiles. For instance, a transaction initially uses a lightweight feedback-driven technique. However, if the size of a transaction (for instance, based on the amount of memory locations read) is above a given threshold, the system switches to a more complex heuristic.

In Nicácio, Baldassin and Araújo (2011, 2012), the *Light-Weight User-Level Transaction Scheduler* (`LUTS`) is proposed. In `LUTS`, each transaction that executes the same critical section shares an identifier (ID). The main idea is to have different scheduling heuristics according to transaction length. The number of cycles is utilized to define if a transaction is long or short. If it is considered short, the heuristic used is similar to `ATS`, using a contention intensity (CI). The difference is that the CI is calculated per

transaction ID, instead of per thread like in the original ATS. Moreover, if a transaction is serialized, LUTS tries to choose one with different ID to replace it. If a transaction is considered long, a more sophisticated (and expensive) heuristic that uses additional data structures is utilized, trying to predict and avoid conflicts. The system uses a fixed thread that is responsible for calculating the length of transactions. Another feature, like in CAR-STM, is to restrict the maximum number of threads to be equal to the cores available.

An extension for LUTS was proposed in (PEREIRA et al., 2013), called BAT (*Best Alternative Transaction*). In LUTS, the CI is utilized for short transactions only. BAT utilize the CI for long transactions too, as an additional parameter for helping to choose the best transaction to be scheduled. In other words, CI is calculated for all transactions. For short ones, it is the unique parameter used to decide which transaction is allowed to run, i.e., with less probability of conflicts. For long transactions, there are more data structures used together with CI to decide it.

The ProVIT scheduler (RITO; CACHOPO, 2015) uses the same idea from LUTS, where there are distinct heuristics according to transaction lengths. However, in ProVIT it is possible to have two heuristics active at the same time, i.e., each thread using a different heuristic. A transaction is considered long if the size of their read-set is greater than a predefined threshold. This information is updated dynamically and, on start, transactions are considered short. For short transactions, the heuristic is based on the previous work of the authors, ProPS, described in Section 3.1.1. When a transaction aborts, the read-set is verified to check if it is a long transaction. If true, it is marked as a VIT (*Very Important Transaction*) and its read-set is copied to a data structure available for all threads. Before a writing transaction commits, it needs to check if its write-set has any intersection with the read-set of any VIT transaction. If it has, the commit should be delayed, giving priority to the VIT. The idea is to avoid a VIT to abort again.

3.1.5 Others

This Section describes works that use machine learning (ML), analytical performance models of applications or other techniques. As these techniques are out of scope of this thesis, they will not be described in details.

Extending the Linux Kernel for supporting STM scheduling was explored in Maldonado et al. (2010, 2011).

The idea of using ML and analytical models in schedulers was widely explored by Rughetti and Sanzo. Controlling the total of threads allowed to concurrently run was proposed in SAC-STM (RUGHETTI et al., 2012), using neural networks, and in CSR-STM (DI SANZO et al., 2013) with analytical performance models. Also, in Rughetti et al. (2014a) a work integrating SAC-STM and CSR-STM, named AML was proposed. Later, Rughetti et al. (2014b) describe an extension to SAC-STM, called DSF-STM. In

Di Sanzo et al. (2016, 2020), a Markov chain-based analytical performance model is proposed to dynamically control the total of concurrent threads executing transactions allowed to run. Castro et al. (2011, 2012) have used ML techniques to choose the best *thread mapping* for improving STM performance. Popovic, Kordic and Basicevic (2017) and Popovic et al. (2019), proposed schedulers for Python-STM (POPOVIC; KORDIC, 2014), using ML techniques.

Finally, Marques and Baldassin (2016), have proposed *Dynamic Serializer* (DS). Different from prior approaches, they have focused on reducing energy consumption instead of performance.

3.2 Thread and data mapping in STM

The works studied so far use transactional scheduling to avoid conflicts between transactions. This section describes works which aim to use scheduler techniques on STM focusing on optimizing resources. For example, trying to keep threads on sibling cores to share caches or load balance in multiprocessor systems.

Castro, Góes and Méhaut (2014) have studied how *thread mapping* could be utilized for improving STM performance. First of all, they describe four different possible thread mappings that could be utilized: Scatter, Compact, Round-Robin and Linux. The proposed mechanism dynamically collects information to decide the mappings. The first strategy was called *Conflict*. It uses the *abort ratio* (AR) as the main heuristic. The intuition utilized is: if AR is high (based on a threshold), the application is accessing a great quantity of shared data. In these cases, using a *Compact* mapping could be useful for sharing caches. If AR is moderated, *Round-Robin*, the intermediate solution, is used; otherwise *Scatter*. The second strategy is called *Test*. It uses the three mappings on a fixed period of time and computes the execution time of each of them. At the end, the mapping that had the minor execution time is selected.

In Chan, Lam and Wang (2015) a thread mapping mechanism called *Affinity-Aware Thread Migration* is proposed. It aims to detect threads that access common data, keeping them in sibling cores on the same processor to share caches. To compute which threads are sharing data, they have used a matrix $n \times n$, where n is the maximum of threads that the system supports. When a thread i conflicts with a j, i.e., aborts, the respective index i, j is updated in the matrix, using the CI (contention intensity) proposed in `ATS` (Section 3.1.1). The intuition used is: if two threads conflict, they were accessing the same shared data. This matrix represents a graph, with edges and their weights. The objective is to reduce the sum of edges between processors. Only a pair is migrated per time.

In Zhou et al. (2018) a concurrency control mechanism to dynamically adjust the total number of active threads executing transactions is proposed. Also, as the total of

threads changes, the authors also adjust thread mapping. The mappings utilized are the same from Castro, Góes and Méhaut (2014), i.e, *Scatter*, *Compact*, *Round-Robin* and *Linux*. The heuristic for deciding the total number of active threads is based on *commit ratio* (CR) and *throughput*. In the beginning, two or more threads (configurable) are allowed to run concurrently. Each period of time, the CR and the throughout are computed, adjusting the total of threads according to it. For thread mapping, the STM system is initialized with the *Linux* default mapping, and after a certain time, it changes to *Round-Robin* (intermediate solution). If the throughput is higher, then *Compact* is used. If performance decreases, then *Scatter* is used.

Góes et al. (2014) focused on STM applications that have a worklist pattern, i.e., they have a single operation responsible to process an item of work from a dynamically managed worklist. Regarding `STAMP` benchmark, four applications present this pattern: *intruder*, *kmeans*, *vacation*, and *yada*. The idea is encapsulated STM application as a skeleton framework, improving the memory affinity by using static page allocation (bind or cyclic) and data prefetching by using helper threads. Helper threads make use of idle cores to bring potentially useful data for the LLC. The memory affinity mechanism is called `SkelAff` and it was implemented inside a proposed `OpenSkel` framework. Hence, STM applications need to be rewritten with this framework to be able to use the memory affinity improvements. It is worth noting this proposed framework is only able to deal with one specific kind of sharing pattern.

3.3 Thread and data mapping in general applications

This section presents works that use thread and/or data mapping to improve data locality of applications that do not use STM, running on shared memory architectures. To identify the works on this subject, the surveys (DIENER et al., 2016a) and (CRUZ; DIENER; NAVAUX, 2018) were used as a basis. We exclude two works described in the papers, because they were out of scope: compiler analysis and fixed runtime options. The former relies on complex analysis that will modify the entire application and not only the STM implementation. Also, this technique usually is limited to specific compiler versions (DIENER et al., 2016a). The latter is used to specify global and static policies, that will be kept fixed until the application finishes, such as interleave for data or round-robin for thread mapping. The remaining techniques will be grouped into two groups: offline and online techniques. Finally, works that focuses on distributed memory, like *Message Passing Interface* (MPI) will not be included, as this thesis does not focus on STM for distributed systems. Although MPI can be used for shared memory architectures, the form of communication is explicit, i.e, tracking the sent messages is possible to discover threads the are communicating often (CRUZ; DIENER; NAVAUX, 2018).

3.3.1 Offline Techniques

This kind of technique consists of analyzing the source code of the application or profiling and executing it, to identify the memory access behavior (DIENER et al., 2016a). Then, the application is modified, either manually or automatically, applying an improved mapping. Behavior analysis happens before execution. At run time, no profile information needs to be collected. However, if the application changes its behavior dynamically, the profile phase will fail on getting precise information about memory access (DIENER et al., 2016a). Profiling could be implemented manually, using system call functions or with the help of instrumentation tools like `Pin` (LUK et al., 2005).

Regarding manually analyzing and source code modifications, some tools could help programmers in this task. `libnuma` (KLEEN, 2004) is a library and command-line tool that is a wrapper for kernel system calls to set memory policies on NUMA architectures. Using `libnuma` as a library, it is possible to allocate memory on a specific NUMA node and the policy to be used, like interleave. Also, it is possible to bind a thread to run on the CPUs of a specific NUMA node. Ribeiro et al. (2009, 2011) have proposed the tools `MAi` (Memory Affinity interface) and `Minas`. `MAi` is a user-level interface focused on numerical scientific HPC applications. It provides optimizations for arrays and memory policies to manage data allocation. The main optimization was made on arrays. When using `MAi` to allocate memory for an array, the programmer can specify the data distribution, i.e., how rows and columns should be distributed in the NUMA nodes. Also, `MAi` implements thread scheduling to ensure data locality. `Minas` is a portable framework for managing memory affinities explicitly or automatically on NUMA architectures. `Minas` has a preprocessor to perform automatic source code modifications and optimizations analyzing the NUMA architecture where the application is compiled. For manual controlling of the memory affinity in the source code, `Minas` uses `MAi`. Another library is `TBB-NUMA` (MAJO; GROSS, 2015; MAJO; GROSS, 2017), that is an extension of the Intel Threading Building Blocks (TBB) (REINDERS, 2007). It includes functionalities for manual management of data allocation between NUMA nodes, automatic thread mapping and other features to make the library NUMA-aware.

Two works use the described tools to achieve better thread and data affinity. Dupros et al. (2010) have studied the impact of memory affinity in seismic simulations. After a detailed study of the source code and the underlying NUMA architecture where the application was executed, `MAi` was utilized for efficient thread and data mapping of the application. Cruz et al. (2011) made similar studies on the NAS Parallel Benchmark (NPB) (JIN; FRUMKIN; YAN, 1999), and after the detailed analysis of the applications, they have used `Minas` for applying the improved thread and data mapping.

There are works that first execute an application to get detailed information about the memory access pattern. Then, this information is used to improve performance. Diener et al. (2010) recorded in a communication matrix which threads have accessed

the same memory location. After that, an algorithm uses this matrix to find the best placement of the threads. If a machine support 16 threads at maximum, then it is feasible to do an exhaustive search, i.e., trying every possible placement. Otherwise, an heuristic is applied. Majo and Gross (2012) have studied the access pattern of scientific benchmarks. The applications were profiled using hardware counters of the Intel Nehalem processor. Also, the source code of the applications were studied in detail, to identify the access patterns of matrices, a common data structure on scientific computation. With this information, the authors have proposed new primitives to be used in OpenMP, to have better control on data distribution and loop iterations over the data structures. Mariano et al. (2016) have made a detailed study in the source code of the HashSieve algorithm, which is used in the context of the lattice-based cryptography technique (MICCIANCIO; REGEV, 2009). The optimization was made by reordering some operations to leverage cache locality, prefetching data and changing data structures used in the algorithm. Denoyelle et al. (2019) have created ML models to choose the best placement of threads and data of applications. They have used different types of thread placement (compact and scatter), data policies (first-touch and interleave) totaling 4 combinations. The application is executed using a default configuration (compact and first-touch) and a set of metrics are collected using instrumentation and hardware counters. Hence, these metrics are processed and used as an input parameter to mathematical and machine learning models. These trained models can identify the best placement for the applications.

Analyzing tools were proposed to collect information about memory access of applications. These tools help to understand how applications share data. `Numalize` (DIENER et al., 2015b) is based on the Pin instrumentation tool (LUK et al., 2005). The authors also proposed metrics to characterize communication and page usage of applications. Thus, `Numalize` generates a memory trace of applications and calculates the proposed metrics. The generated information helps to choose the best placement of threads and data. `TABARNAC` (BENIAMINE et al., 2015) is similar to `Numalize` and provides graphical visualization of memory access behavior, like the distribution of the accesses by threads and data structures. A most recent tool, `NumaMMa` (NUMA MeMory Analyzer) (TRAHAY et al., 2018) uses hardware counters of a processor to generate memory traces. When the application finishes, `NumaMMa` processes the trace and analyzes the cost of memory accesses of each object and how threads access them. Also, graphical visualization of the processed information is available.

Denoyelle et al. (2019), obtain information about the characteristics of applications through a preliminary execution. This information is collected using instrumentation or hardware performance counters. After that, using machine learning, the collected information is processed and, resulting in information if application is sensitive to locality and the best thread and data placement. Then, the application is re-executed using the

proposed placement.

3.3.2 Online Techniques

Online techniques collect information about memory access during the execution of the application and perform mappings dynamically. The main advantage is that no prior execution is needed to collect information. However, the main challenge is to create an heuristic that takes into consideration the trade-off between accuracy and runtime overhead (DIENER et al., 2016a). Analyzing all memory accesses of an application is the best strategy for deciding mappings, but the overhead added makes it unfeasible (CRUZ; DIENER; NAVAUX, 2018). Thus, usually, the methods used to collect information about memory accesses is based on sampling. Also, the migration of threads and pages represent an overhead that should be considered.

Ogasawara (2009) have used the Garbage Collector (GC) of the Java programming language to identify the preferred NUMA node for objects. During the collection phase, the proposed method determines which threads have most accessed certain objects, i.e., the *Dominant Thread* (DoT). Then, the GC migrates the objects to the NUMA nodes where the DoT is running.

`ForestGOMP` (BROQUEDIS et al., 2010b) is an extension of OpenMP that uses hints provided by application programmers, compiler techniques and hardware counters to perform dynamically thread and memory placement. It uses internally libraries like `hwloc` to compute the underlying hardware hierarchy and the `BubbleScheduler` framework (THIBAULT; NAMYST; WACRENIER, 2007). Threads that share data or synchronize often are organized in bubbles. The main objective of the proposed scheduler used by `ForestGOMP` is to improve the cache usage by making each “bubble” access the local memory, migrating pages if necessary. The scheduling actions are triggered when resources are (de)allocated, a processor becomes idle or a hardware counter exceeds a predefined threshold.

`autopin` (KLUG et al., 2011) is a framework that extends the `pfmon` utility from the `perfmon2` package (ERANIAN, 2005). `pfmon` is a command line tool that accesses hardware performance counters of processors. Through an environment variable it is possible to choose a set of mappings, i.e., the cores to be used and the order that they should be assigned to threads. A command-line parameter is available for setting a hardware counter that `autopin` should use for calculating the performance rate. After the initialization, if more than one set of mappings was defined, `autopin` will test each one, calculating the performance based on the timestamp and the hardware counter chosen. After all mappings have been tested, the one which achieves the best performance will be chosen and will be active until the application finishes.

The focus of the `BlackBox` scheduler (RADOJKOVIĆ et al., 2013) is applications with a low number of threads and many instances, like networking applications. It tries to

test all possible thread mappings (assignments) and chooses the one that achieves the best performance. It has 3 distinct phases. The first one profiles the target application. The main information collected are conflicts and shared resources, like caches and memory controllers. The second phase uses the information collected to predict the performance of a determined thread assignment. If the number of threads and instances are low, all possibilities are tested. Otherwise, only 1000 combinations are evaluated. The last phase chooses the best 5 predicted models and tests them to make sure that the best one was chosen.

Tam, Azimi and Stumm (2007) have used hardware performance counters of the IBM Power5 processor to track cache misses between chips, i.e., if the miss was local or remote. Each thread has a data structure to keep track of data regions accessed that caused a remote cache miss. This information is retrieved from a counter in the processor. Analyzing all cache misses would be unfeasible, thus, the authors sampled the monitoring. The next step is to analyze data structures looking for data sharing between threads. If found, threads are mapped to the same chip, for sharing caches. After that, the monitoring phase starts again. The proposed scheduler was tested with a synthetic micro-benchmark and three commercial server workloads. In Azimi et al. (2009) the authors have extended the studies of how the hardware performance counters could be utilized for improving locality of threads. They have presented the usability and importance of hardware performance counters as a low-overhead resource to get accurate online information about performance.

Regarding data mapping, Löf and Holmgren (2005) have studied how a next-touch directive could improve the performance of industrial applications. One application was chosen that uses a large matrix initialized by the main thread. After the initialization, it consumes about 500 MB of memory. Using the default Linux OS data mapping, first-touch, all pages will be allocated in the first node. With next-touch, the data will be migrated according to the accesses of threads. The author has identified a great overhead on page migrations, due to TLB (translation look-aside buffer) shootdowns, i.e., the mechanism that keeps the TLB's coherent (VILLAVIEJA et al., 2011). To overcome this limitation, they have proposed to use larger pages, e.g., 64Kb instead of 8Kb.

The *Communication Detection in Shared Memory* (CDSM) (DIENER et al., 2015a) is a Linux kernel module to detect communication patterns of parallel applications and migrate threads according to their memory affinity. It keeps track of page faults to detect the communication between threads. The *kernel Memory Affinity Framework* (kMAF) (DIENER et al., 2016c) is implemented directly in the Linux kernel, extending the idea of CDSM to the problem of data mapping in NUMA architectures. The *Carrefour* mechanism (DASHTI et al., 2013) is also implemented directly in the Linux kernel. However, it uses Instruction-Based Sampling (IBS) available only in AMD processors.

Barrera et al. (2020) have the same objective of Denoyelle et al. (2019) (described in Section 3.3.1), however their model works online, and they have added prefetching optimizations beyond thread and data placement.

3.4 Workload characterization

In Barrow-Williams, Fensch and Moore (2009), the Splash-2 and Parsec benchmark suites were characterized regarding their memory access behavior. They also showed other characteristics such as temporal and spatial characterization of the communication. All information was gathered through the use of a simulator. Using the collected information, they proposed changes to the communication systems to be used in future chip multiprocessors (CMP). Different communication characteristics of Splash-2 and Parsec were also studied in (MOHAMMED; ABANDAH, 2015). Rane and Browne (2011) proposed to instrument the source code of applications using the LLVM compiler, to extract memory trace of applications. Then, they compute metrics using the collected data, such as cycles per access; NUMA hit ratios; access strides, etc. With this information, they manually changed a subset of programs of the Rodinia benchmark suite and were able to improve the performance. The `numalize` tool to extract information about communication behavior was proposed in (DIENER et al., 2015b). Also, they have analyzed different benchmark suites and proposed metrics that describes spatial, temporal, and volume properties of memory accesses to shared areas.

Hughes et al. (2009) proposed a set of transactional characteristics to classify the similarity of transactional workloads. The main idea is to select a subset of programs that present distinct transactional characteristics, to be used in benchmarks or tests. In (CASTRO et al., 2011), a generic mechanism was proposed to intercept any function of STM libraries. The mechanism is implemented as an external tool, allowing developers to extract and calculate information accessed inside the STM library. Baldassin, Borin and Araujo (2015) characterized the memory allocation of `STAMP`.

Other studies aim to characterize the energy consumption of STM workloads. Baldassin et al. (2012) studied the difference of energy consumption of the `STAMP` benchmark using three STM libraries, along with three different conflict resolution scheme. This analysis was made using a simulator. As a result of the studies, they have proposed to integrate a *dynamic voltage and frequency scaling* (DVFS) inside the contention manager, to improve the energy efficiency and performance. Rico et al. (2015) also studied the energy consumption of STM using the `STAMP` benchmark. However, the study was made using a real machine, instead of a simulator.

Table 2 – Comparison of related work on thread and data mapping for STM applications.

Work	Year	Thread mapping	Data Mapping	Sharing based
Castro, Góes and Méhaut	2014	X		
Góes et al.	2014	X	P	
Chan, Lam and Wang	2015	X		P
Zhou et al.	2018	X		
This thesis	2021	X	P	X

3.5 Discussion

As described in Section 3.1 a transactional scheduler is a well-known technique for improving the performance of STM. The majority of described transactional schedulers focuses on serializing transactions to avoid conflicts. They rely on the idea that aborting a transaction is a waste of time and resources, and it is necessary to avoid it. Although avoiding aborts in an STM execution is also important for improving performance, in current multicore architectures with complex memory hierarchies and different latencies, it is also important to consider where the memory of a program is allocated and how it is accessed. Hence, our idea is to work with thread mapping in STM, investigating the use of sharing-aware mapping (Section 2.3) in an STM implementation. This technique aims to map threads to cores and memory pages to NUMA nodes considering their memory access behavior.

Table 2 makes a direct comparison of our proposed work with the related work on thread and data mapping for STM (Section 3.2). The works of Castro, Góes and Méhaut (2014) and Zhou et al. (2018) do not take into consideration memory access behavior to decide the thread mapping (column *Sharing based*). Góes et al. (2014) focused on a specific sharing pattern (*worklist*) and their mechanism to exploit memory affinity were implemented in a framework. Hence, applications need to be rewritten with this framework to be able to use the memory affinity improvements. Also, their data mapping is based on static page allocation. Thus, we classified data mapping support in their work as partial (*P*) in Table 2. We propose not to rely on a specific sharing pattern, as we intend to calculate the thread mapping based on the sharing pattern, stored in a communication matrix (Section 2.3.1). Also, it would not be necessary to rewrite the STM application, only recompile it, since the mechanism for thread mapping will be integrated into the respective STM runtime. The main focus of this thesis is on thread mapping. However, we show how data mapping can be added to the proposed mechanism of sharing-aware thread mapping. Hence, we classified our data mapping support as partial (*P*). Finally, Chan, Lam and Wang (2015) take into consideration the memory access behavior of applications. However, they proposed to compute the sharing behavior only when a transaction aborts. The intuition is that if two transactions conflict, they were accessing the same shared variable. The disadvantage is that their

mechanism does not capture the sharing behavior between read-only transactions. Besides, in low contention workloads, applications present a low number of aborts. Therefore, their mechanism could not capture an accurate sharing behavior of these STM applications. Our idea is to compute the sharing behavior using transactional reads and writes, making it more accurate. Furthermore, our proposed mapping is global, i.e., taking into consideration all threads, not only one pair each time.

Concerning the characterization of STM applications regarding sharing behavior, Section 3.4 showed that so far, no works in the literature were proposed with this objective.

3.6 Summary

This chapter presented the related work on the thesis subject. There are several works on transactional schedulers. Although reducing the number of aborts improves the performance, recent works use schedulers (thread and data mapping) with the objective of improve cache usage, page sharing or reduce the latency on data access. However, no work on STM applications takes into consideration memory access behavior to performing thread and data mapping. Besides, this chapter showed that no related work performed a characterization of STM applications regarding sharing behavior. The next chapters will explore these research opportunities to advance the state-of-the-start on this subject.

4 DETECTING MEMORY ACCESS BEHAVIOR IN STM APPLICATIONS

In this chapter, a mechanism to detect the sharing behavior of STM applications is proposed. It is the base mechanism used in all thesis contributions.

4.1 Overview

Detecting the memory access behavior is the first step to perform a sharing-aware thread mapping. It is necessary to know which memory address is being accessed and which thread is accessing it. In shared memory programming models, communication is implicit, i.e., it is performed through access to shared memory areas. This characteristic adds more challenge to detect the memory access behavior in shared memory architectures.

Detecting the communication behavior accurately with tools such as `numalize` (DIENER et al., 2015b) cause high overheads. Hence, it necessary to use a heuristic to detect communication events accurately and with low overhead. A *communication event* happens when at least 2 threads access the same shared variable. Although writes to memory are more expensive, i.e., they imply in cache invalidation and interconnection traffic between remote notes, we will not differentiate reads from writes in the proposed mechanism. Besides, we will take into consideration the operations performed by all transactions, including aborted. If one transaction aborts, it is accessing shared variables with other transactions. If we take into consideration only committed transactions, this relationship would not be registered, making the mechanism less precise. Also, it is important to have a heuristic to avoid false temporal communication, i.e., two threads access the same shared variable but in long difference execution intervals. In that case, during the second access, the first one already has been evicted from the cache lines.

To store the amount of communication between pairs of threads, we use a communication matrix (Section 2.3.1), where each matrix position represents the amount of communication between pairs of threads. Since access to a shared variable from the same thread does not represent a communication event, matrix diagonals are zero.

4.2 Mechanism

The main idea of this mechanism is to work inside the STM library, by only keeping track of STM accesses. To detect the memory access behavior of an application, it is necessary to know which addresses the application is accessing. In STM runtimes, each transactional data access operation explicitly includes the addresses used in the operation. For instance (HARRIS; LARUS; RAJWAR, 2010):

```
T ReadTx(T *address);
void WriteTx(T *address, T value);
```

As this is already available to the STM runtime, it is not necessary to rely on external tools or add instrumentation overhead to get this information. Besides, the STM runtime knows which thread is performing the access. Since STM runtimes have precise information about shared variables, it is possible to determine the communication behavior by tracking transactional reads and writes instead of tracking all memory accesses. Our detection mechanism works as follows. When at least 2 distinct threads access the same address, a communication event between them is updated in a communication matrix.

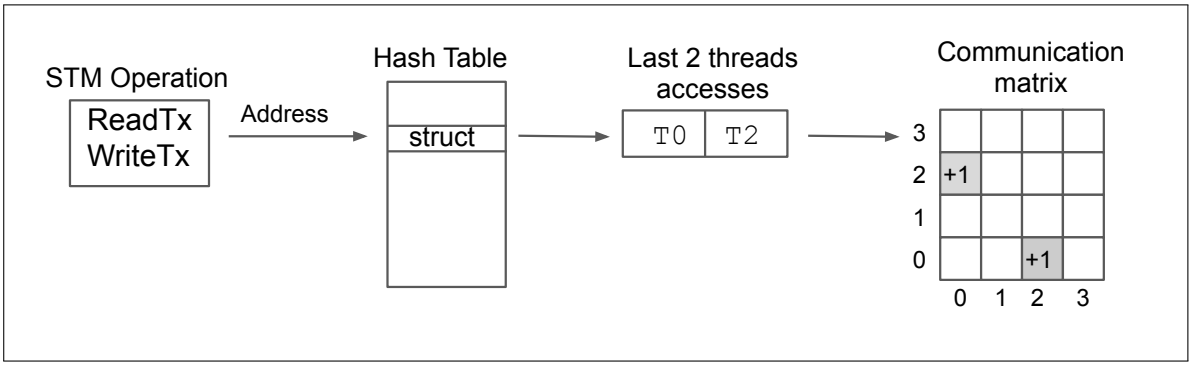


Figure 5 – Mechanism for detecting communication patterns. Data structures are shown for an application consisting of 4 threads (0-3.)

Our mechanism to detect the communication pattern of an application is shown in Figures 5–6 and detailed in Algorithm 2. This mechanism is executed before each transactional read or write operation. To keep track of accessed addresses, a hash table is used, in which keys are memory addresses. Each position of the hash table contains a structure with the memory address and the last 2 threads that have accessed it. Following the intuition of previous work, we store only the last 2 thread access, to avoid false temporal communication (DIENER et al., 2016b). In case of hash conflicts, a linked list with all memory addresses with the same hash is kept. In line 1, the function `getAddressInfo` gets from the hash table the structure containing information about the address being accessed. The next step is to get how many accesses this address had before the current one (line 2). If it is the first access (line 3), we only store the thread that is accessing it (line 4), i.e., no communication events have happened yet.

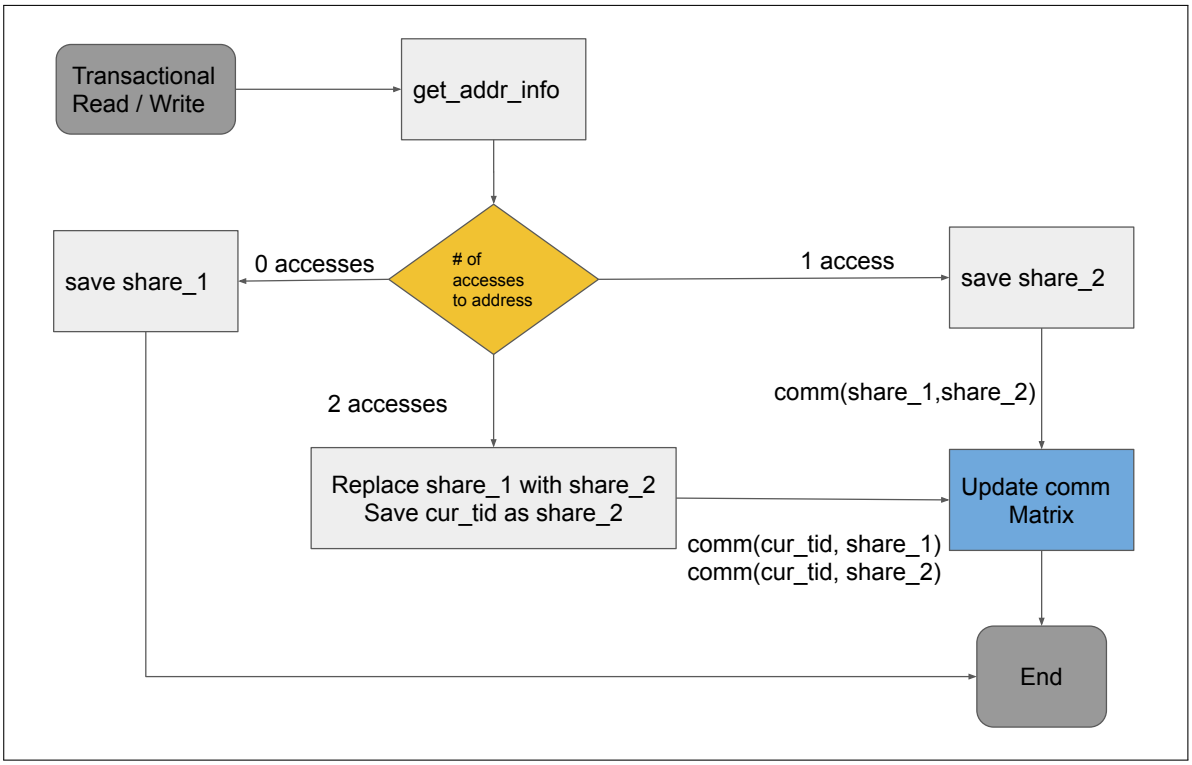


Figure 6 – Flowchart of the proposed mechanism.

The next case is if the address had one access before the current (line 5) and it was made by a different thread (line 6). If true, we have a communication event. In that case, we call the function `update_comm_matrix` (line 7) to update the communication matrix, increasing the amount of communication between the two threads. The matrix is square and symmetric because the amount of communication, for instance, between threads 0 and 2 is the same from 2 and 0 (Figure 5). Also, the matrix has an order equal to the maximum number of threads. Using the matrix, we update the threads that accessed the address (lines 8 and 9). It is worth noting that write accesses to the communication matrix are not synchronized. This decision was made because the high overhead involved to synchronize the access. Besides, the collateral effects expected due to the parallel updates on the matrix, for instance, incorrect pairs of thread being incorrectly updated in the matrix, are considered eventual and not harmful to the mechanism.

The final case is if the address had 2 previous accesses (line 10), then there are 3 sub-cases. The first one is if the current thread accessing the address is different from the 2 previous (line 11). In that case, we have a third distinct thread accessing the address, and we update the communication matrix between the 3 (lines 12 and 13). After that, the oldest access is removed (line 14). However, if the test of the line 11 was false, it means that the current thread accessing the address is the same from a previous one. In that case, we need to check which thread is accessing again (lines 16 and 18), and update the communication matrix correctly. Also, since we keep only the

Algorithm 2 Detecting communication patterns.

Require:
addr: memory address being accessed
tid: thread ID of the thread that is accessing the address

▷ `elem` is a struct that stores the memory address and the last 2 threads that have accessed it
1: $elem \leftarrow$ getAddressInfo$(addr)$
2: $accesses \leftarrow$ getAccesses$(elem)$
3: **if** $(accesses = 0)$ **then** ▷ First access to the address
4: $elem.t1 \leftarrow tid$
5: **else if** $(accesses = 1)$ **then** ▷ One previous access
6: **if** $(elem.t1 \neq tid)$ **then**
7: update_comm_matrix$(tid,\ elem.t1)$
8: $elem.t2 \leftarrow elem.t1$
9: $elem.t1 \leftarrow tId$
10: **else if** $(accesses = 2)$ **then** ▷ Two previous accesses
11: **if** $(elem.t1 \neq tid)$ **and** $(elem.t2 \neq tid)$ **then**
12: update_comm_matrix$(tid,\ elem.t1)$
13: update_comm_matrix$(tid,\ elem.t2)$
14: $elem.t2 \leftarrow elem.t1$
15: $elem.t1 \leftarrow tid$
16: **else if** $(elem.t1 = tid)$ **then**
17: update_comm_matrix$(tid,\ elem.t2)$
18: **else if** $(elem.t2 = tid)$ **then**
19: update_comm_matrix$(tid,\ elem.t1)$
20: $elem.t2 \leftarrow elem.t1$
21: $elem.t1 \leftarrow tid$

last 2 accesses to the memory address, when a third thread access the address, it is necessary to replace the oldest one (line 20).

4.3 Implementation

We implemented our proposed mechanism inside the state-of-art STM library `TinySTM` (FELBER et al., 2010), version 1.0.5. The majority of the modifications were made in the file `stm_internal.h`. The Algorithm 2 is called inside the functions `stm_write` and `stm_load` from `TinySTM`.

4.4 Evaluation

This section describes experiments made in order to evaluate the proposed mechanism to detect the sharing behavior of STM applications.

Table 3 – Default arguments for the programs used in the experiments.

Application	Arguments
bayes	`-v32 -r8192 -n10 -p40 -i2 -e8 -s1 -t num_threads`
genome	`-g49152 -s256 -n33554432 -t num_threads`
intruder	`-a10 -l128 -n262144 -s1 -t num_threads`
kmeans	`-m40 -n40 -t0.00001 -i random-n65536-d32-c16.txt -p num_threads`
labyrinth	`-i random-x1024-y1024-z9-n1024.txt -t num_threads`
ssca2	`-s21 -i1.0 -u1.0 -l3 -p3 -t num_threads`
vacation	`-n4 -q90 -u100 -r1310720 -t16777216 -c num_threads`
yada	`-a15 -i ttimeu1000000.2 -t num_threads`
redblacktree	`-u 20 -i 1000000 -d 500000000 -n num_threads`
hashmap	`-u 20 -i 1000000 -d 500000000 -n num_threads`

4.4.1 Methodology

We used the proposed mechanism to collect the communication matrices of STM applications by using the modified version of the `TinySTM` library (Section 4.3), with the default options: *lazy* version management, *eager* conflict detection and contention manager *suicide*.

The applications used in the experiments were all eight benchmarks from the Stanford Transactional Applications for Multi-Processing (`STAMP`) (MINH et al., 2008) version 0.9.10, and two micro-benchmarks (HashMap and Redblacktree) from Diegues, Romano and Rodrigues (2014). The input arguments used to run each application are shown in Table 3. Most parameters are larger than the largest ones suggested in the original paper (MINH et al., 2008) to achieve more substantial execution times on modern machines. To run the experiments, we used the the following NUMA machine (node distances were gathered with `numactl` (KLEEN, 2004)):

- **Xeon**: 8 Intel Xeon E5-4650 processors and 488 GiB of RAM running Linux kernel 4.19.0-9. Each CPU has 12 2-HT cores, totaling 96 cores. Each CPU corresponds to a NUMA node (for a total of 8 NUMA nodes), and 12$\times$ 32 KB L1d, 12$\times$ 32 KB L1i, 12$\times$ 256 KB L2 and 30 MB L3 cache. Node distances: 50 – 65. Applications were compiled using gcc 8.3.0.

4.4.2 Communication matrices

Figures 7–10 shown the collected communication matrices using the proposed mechanism for 16, 32, 64 and 96 threads. In the figures, axes show threads IDs and darker cells indicate more communication between pairs of threads.

The two micro-benchmarks (HashMap and Redblacktree) present a communication pattern where neighbor threads communicate often. In that case, darker cells are localized closer to the main diagonal of the matrix. For this pattern, it is interesting to map

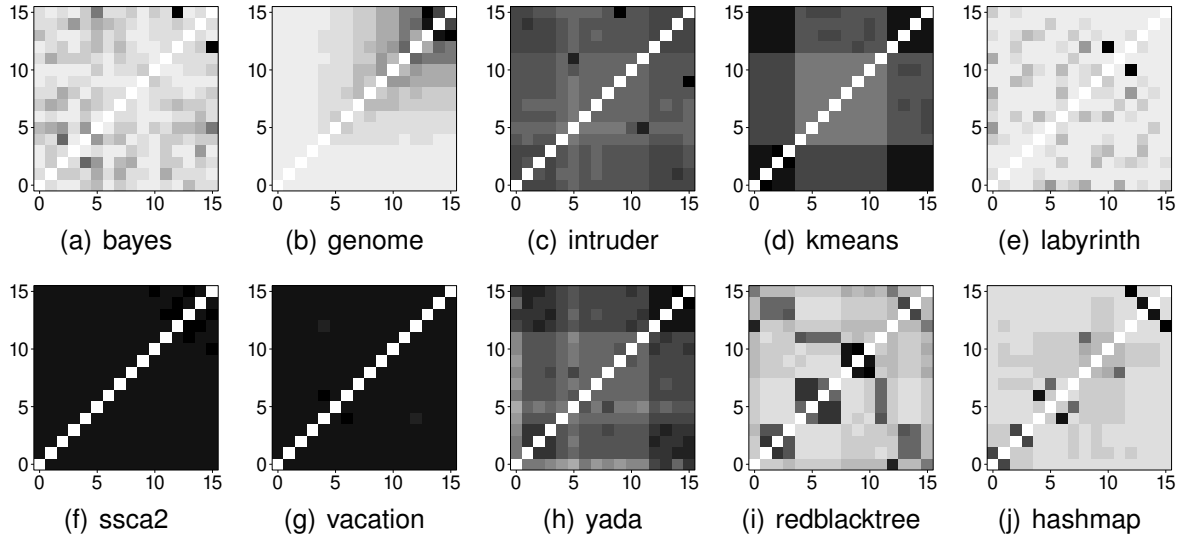


Figure 7 – Communication matrices - 16 threads.

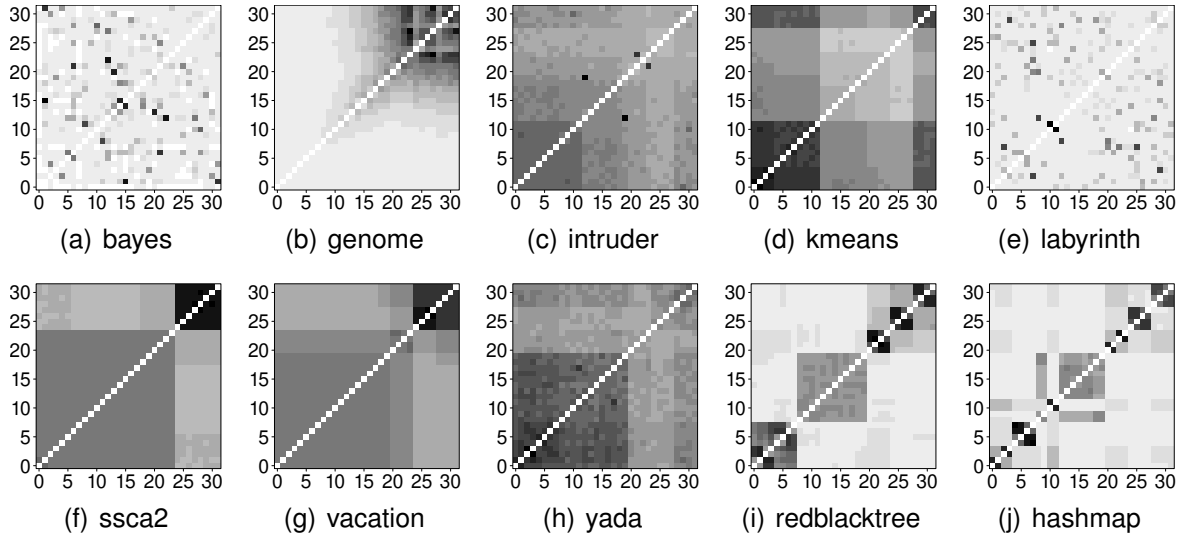


Figure 8 – Communication matrices - 32 threads.

threads on sibling cores to share all cache levels. On *Genome*, only threads with higher IDs communicates often. Applications such as *ssca2* and *vacation* have an intense communication with all threads, know as all-to-all pattern (BARROW-WILLIAMS; FENSCH; MOORE, 2009). On the other hand, *labyrinth* and *bayes* present low communication intensity, but they are considered as an all-to-all pattern as well.

4.4.3 Overhead

To compare the overhead generated by tracking and generating the communication matrices, we compare our mechanism with a memory tracing tool called `numalize` (DIENER et al., 2015b). For some applications `numalize` crashes and it is not possible the extract the communication matrix. This problem was also observed in others studies that have used this tool (SOOMRO; SASONGKO; UNAT, 2018): "*Unfortunately, in some*

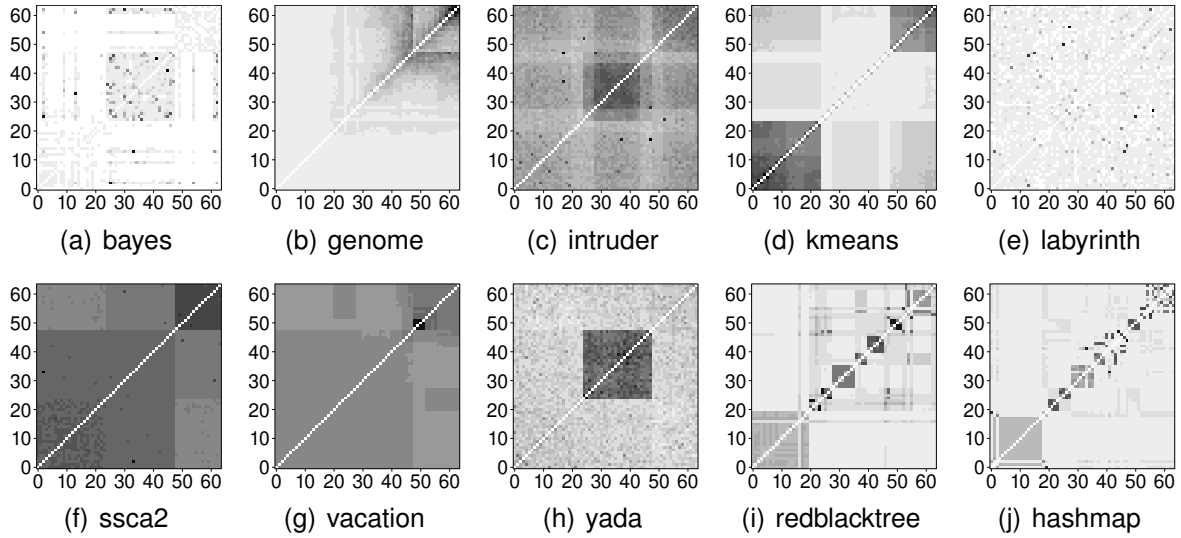


Figure 9 – Communication matrices - 64 threads.

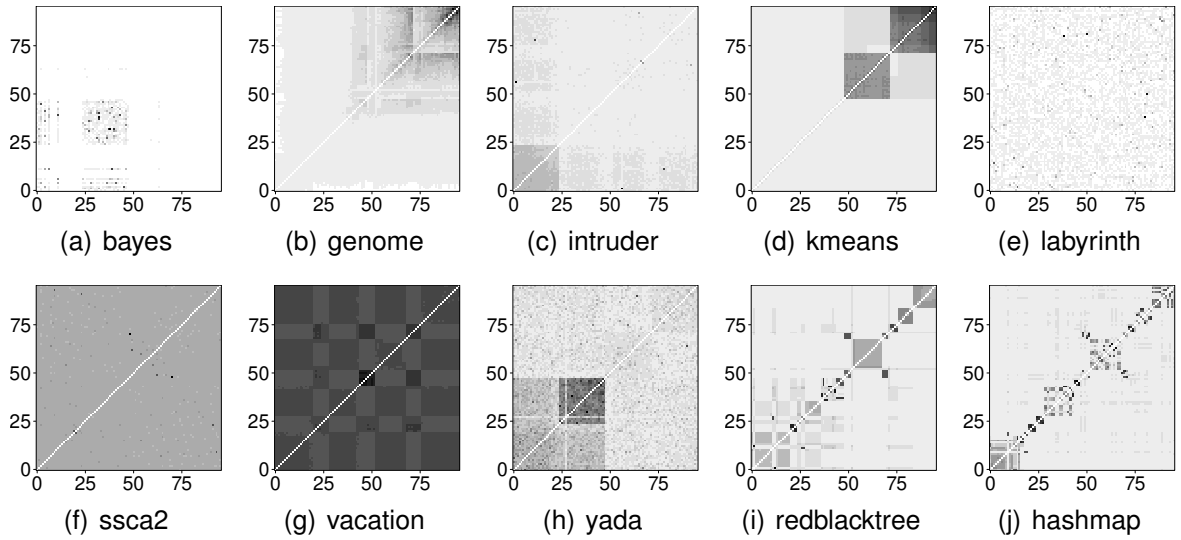


Figure 10 – Communication matrices - 96 threads.

cases, Numalize crashes because of the large memory requirements of an application in addition to its own internal data structures". Hence, we were only able to run *kmeans* using `numalize`.

`Numalize` depends on Intel's `Pin` tool (LUK et al., 2005) to instrument the application and trace all accessed addresses, not only the ones accessed by the STM system. Therefore, `numalize` captures a different memory access behavior compared to our mechanism. This is visualized in Figure 11 which compares the collected matrices of *kmeans* with 32 and 64 threads using both mechanisms.

Another disadvantage of `numalize` is the overhead added to trace all memory accesses. On *kmeans*, using 64 threads, `numalize` took 690.90 seconds to execute the application and collect the communication matrix. By contrast, our mechanism took only 46.20 seconds for the same operation, almost 15$\times$ less. The normal execution

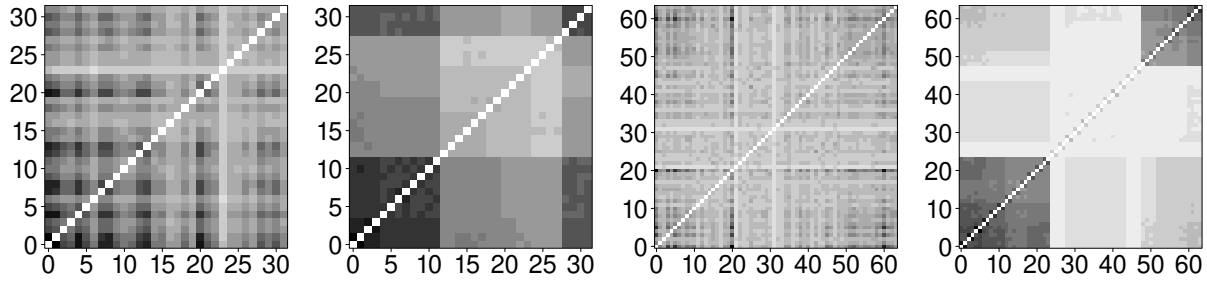


Figure 11 – Comparing `numalize` and our mechanism on *kmeans*.

time for *kmeans* without tracing anything is 18.05 seconds, such that `numalize` added a overhead of 38.27×, whereas the overhead was 2.5× with our mechanism. Although the overhead of our mechanism can be considered low, it is unfeasible to be used in an online mechanism. In Section 6.2.1, we will show how to reduce this overhead to be able to perform the detection of sharing behavior during runtime.

4.5 Summary

This chapter presented a mechanism to detect the sharing behavior of STM applications. Since STM runtimes need the memory address on each data access operation and have precise information about shared variables, it is possible to determine the communication behavior by tracking transactional reads and writes instead of all memory accesses. Using the proposed mechanism it was possible to extract the sharing behavior of STM applications with lower overhead than other memory trace tools, such as `numalize`. Although the proposed mechanism has lower overhead, additional experiments are necessary to verify if the collected information is accurate, for instance, if they can be used to calculate an efficient thread mapping. These experiments will be made in Chapter 6.

5 CHARACTERIZATION OF SHARING-BEHAVIOR OF STM APPLICATIONS

For a successful thread mapping, it is necessary to perform an in-depth analysis of STM applications, for instance, if the memory access pattern changes in each execution; the number of addresses accessed inside transactions, etc. This analysis is important to guide decisions regarding mapping, such as determining if an application is suitable for a thread mapping based on communication behavior and defining the type of mapping policy (static or dynamic).

In this chapter, we characterize the applications from STAMP (MINH et al., 2008), by gathering sharing information through the proposed mechanism in Chapter 4, providing information to guide thread placement based on their sharing behavior. The STAMP benchmark was developed with the intention to represent realistic workload characteristics and different application domains. Besides, STAMP covers a wide range of transactional behavior, for instance, varying transaction lengths, contention, quantity of transactions, etc. Hence, we expect that the characterization done in this chapter could represent a wide range of real STM applications. Using the proposed mechanism we are able to gather information about the suitability for thread mapping of each application, its communication pattern, and its dynamic behavior, among others. We also show how this mechanism can be used to detect false sharing of cache lines of STM operations.

5.1 Methodology of the Characterization

5.1.1 Detecting sharing in STM applications

The characteristics of memory access behavior presented in this thesis are based on the communication matrices of applications. To extract the matrices we have used the mechanism proposed in Chapter 4. STAMP was compiled with gcc 8.3.0. If not specified otherwise, all applications were executed ten times using 64 threads and the default input parameters shown in Table 3. We also used the same Xeon machine described on Section 4.4.1 to run the experiments.

5.1.2 Mean squared error (MSE)

Since the analysis of the communication behavior of the applications is based on communication matrices, we used *Mean squared error* (MSE) (WANG; BOVIK, 2009) metric to compare the difference between them, which has been used in prior work for this purpose (DIENER et al., 2016b). The equation to calculate the MSE is shown in Eq. 1.

$$MSE(A,B) = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (A[i,j] - B[i,j])^2 \quad (1)$$

where:

$A, B =$ input matrices
$N \quad =$ matrix order, i.e., number of threads
$i, j \quad =$ matrix indexes

If the MSE of two matrices is zero, then the matrices are exactly the same. Higher MSE values indicate higher differences. This metric is useful to compare, for instance, if the memory access behavior of one application changes on each execution.

5.1.3 Experiments

We performed the following experiments to characterize the sharing behavior of the `STAMP` applications:

1. We collected information about the total of accessed addresses inside the STM library. The information is useful to understand how much data is accessed by STM operations (Section 5.2.1).

2. We executed the same application ten times to verify if the communication pattern changes between executions. This experiment will show if a communication matrix collected in a previous execution can be used to make a static thread mapping (Section 5.2.2).

3. We executed the same application ten times, changing the input parameters from the default. This experiment will show if it is possible to use a collected communication matrix with different input parameters to make a static thread mapping (Section 5.2.3).

4. We executed the same application ten times, changing the total number of threads from the previous experiments, with the same goal as in the previous item (Section 5.2.4).

5. We collected the communication matrix several times during the execution of an application, to determine if an application needs an online mechanism to detect the sharing behavior and perform the thread mapping multiple times during execution (Section 5.2.5).

5.2 Characterization of sharing behavior

This section presents the characterization of the `STAMP` applications, regarding memory access behavior.

5.2.1 STM memory access information

The first data set is not directly related to sharing behavior but is useful to explain further the behavior of applications. The total number of distinct memory addresses accessed by STM operations was collected as well as the total number of accesses made to these addresses such as read or write operations. With this data, it is possible to calculate other information:

- **Number of distinct cache lines**: This was calculated based on the default cache line of most current microarchitectures, 64 bytes.
- **Number of distinct pages**: This was calculated based on the default page size of many current microarchitectures, 4096 bytes.
- **Percentage of cache lines with false sharing**: We consider that a cache line has false sharing when multiple threads perform STM operations on more than one word at the same line.

Table 4 – Analysis of accessed STM memory addresses in STAMP applications.

Application	Distinct addresses	Distinct cache lines	Distinct pages	Total accesses	% of lines with false sharing
bayes	1,082	497	122	15,928,303	89.33
kmeans	682	101	4	833,954,131	100.00
labyrinth	824,126	290,083	18,213	1,994,315	83.05
genome	19,750,104	11,216,870	452,651	2,840,508,725	36.21
intruder	23,292,164	8,131,906	297,629	4,105,619,590	99.00
yada	25,055,077	13,932,226	848,045	610,446,722	81.38
vacation	33,671,744	10,796,118	799,129	7,212,365,036	99.08
ssca2	91,335,091	13,907,852	528,387	270,369,505	99.90

Results are shown in Table 4 and indicate that the applications have different characteristics regarding the number of accessed addresses. *kmeans* has the lowest amount of distinct addresses accessed. However, it has a large number of total accesses made

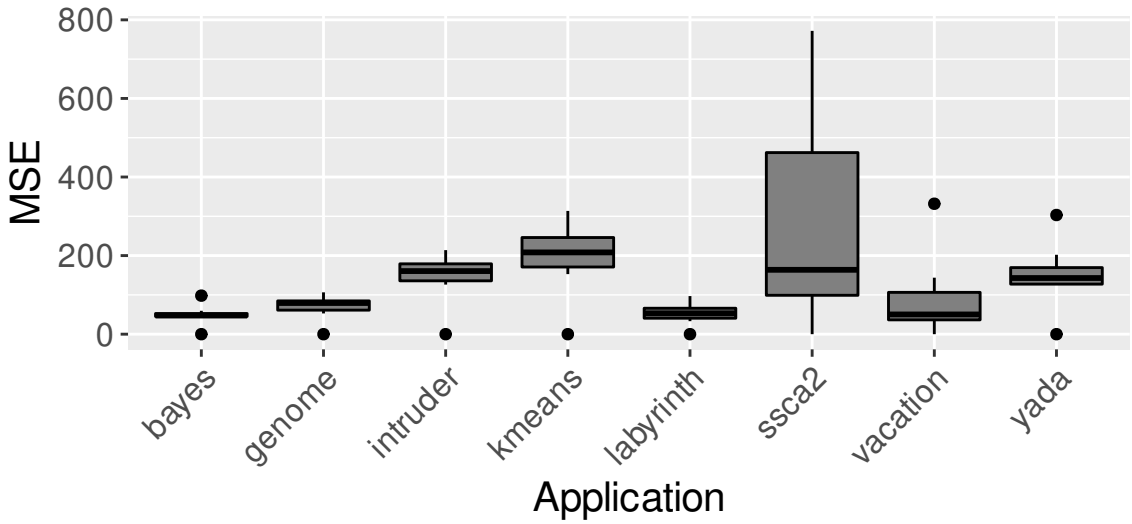


Figure 12 – Stability of the sharing behavior across different executions.

to these addresses. On the other hand, *ssca2* has the largest amount of distinct addresses accessed, but not the largest total of accesses, which belongs to *vacation*. Regarding false sharing, the majority of applications have a large percentage of false sharing. For instance, in *kmeans*, all addresses share common cache lines. On the other hand, *genome* has the least amount of false sharing which indicates that more than half of the accessed addresses has at least 64 bytes or the addresses that conflict in the same cache line are accessed outside the STM library.

5.2.2 Stability of sharing behavior across different executions

The goal of this experiment is to determine if the communication pattern changes across different executions of the same application, using the same input parameters and number of threads. To answer this question, we executed all applications one time and collected the communication matrix, to be used as a baseline for comparison. After that, we run each application nine more times, collecting the communication matrix in each execution. Then, we calculated the MSE of each resulting matrix, comparing it to the first execution. Results are shown in Figure 12.

Some applications, for instance *bayes*, *genome* and *labyrinth*, present the same communication behavior in all executions. This observation can be visualized in two different communication matrices of *bayes* (Figure 13(a) and Figure 13(b)). Axes show threads IDs. In contrast to *bayes*, *ssca2* presents a not so similar behavior on each execution. However, looking at two *ssca2* matrices (Figure 13(c)) and Figure 13(d)) it is possible to note that although the basic communication behavior is the same (all-to-all (BARROW-WILLIAMS; FENSCH; MOORE, 2009)), the total amount of communication between threads is very different. This can be explained by the non-deterministic behavior of TM applications, mainly due to the fact that the total number of aborts varies in each execution. More aborts imply in more work to be done, consequently more

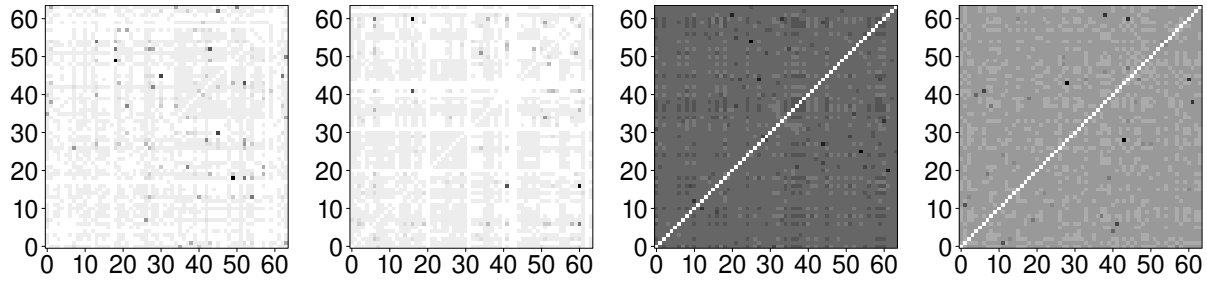


Figure 13 – Matrices with highest and lowest MSEs between different executions.

communication between threads. In that case, even having a higher MSE between executions, *ssca2* has a similar behavior of communication between threads (all-to-all pattern) in all executions.

5.2.3 Stability of sharing behavior when changing input parameters

For this experiment, instead of using the default input parameters shown in Table 3, we used a smaller input data set. The changed parameters are shown in Table 5. Then,

Table 5 – Small input parameters used in the experiments in Section 5.2.3.

Application	Arguments
bayes	`-v16 -r4096 -n15 -p40 -i2 -e8 -s1 -t num_threads`
genome	`-g16384 -s64 -n16777216 -t num_threads`
intruder	`-a10 -l64 -n131072 -s1 -t num_threads`
kmeans	`-m15 -n15 -t0.00001 -i random-n65536-d32-c16.txt -p num_threads`
labyrinth	`-i random-x512-y512-z7-n512.txt -t num_threads`
ssca2	`-s18 -i1.0 -u1.0 -l3 -p3 -t num_threads`
vacation	`-n4 -q60 -u90 -r1048576 -t4194304 -c num_threads`
yada	`-a20 -i ttimeu100000.2 -t num_threads`

we collected ten communication matrices using the same methodology of Section 5.2.2. Lastly, a comparison of the MSE using the default parameters (Section 5.2.2) was made, comparing with the small parameters (Table 5). This comparison is shown in Figure 14.

As in the previous experiment, *ssca2* has a different pattern on each execution. For instance, Figure 15(c) and Figure 15(d) show two different executions of *ssca2*, using the small parameters (Table 5). However, with the new parameters, it is possible to observe that some groups of threads communicate more often than others (Figure 15(c)). Besides, there is a difference between communication patterns taking into consideration the default and small parameter sets. This can be visualized by comparing Figure 13(c) and Figure 15(c). Other applications such as *intruder*, *kmeans*, and *vacation* have a small difference between communication patterns when changing input parameters. While others, such as *genome* have almost the same communication pattern, even

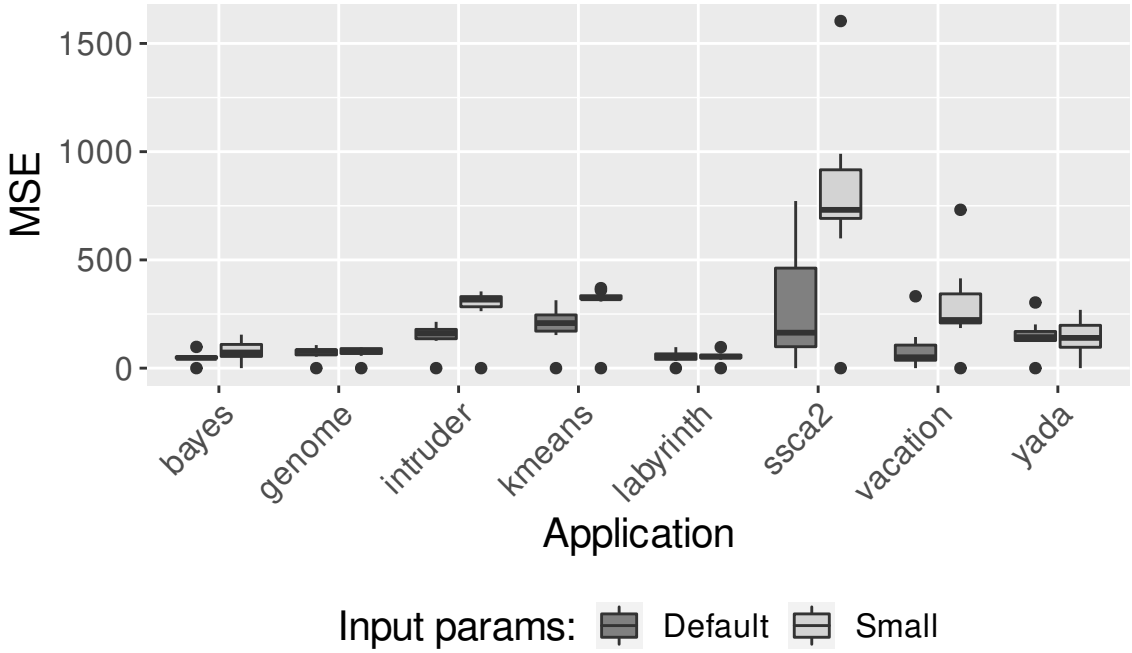


Figure 14 – Stability of the sharing behavior when changing input parameters.

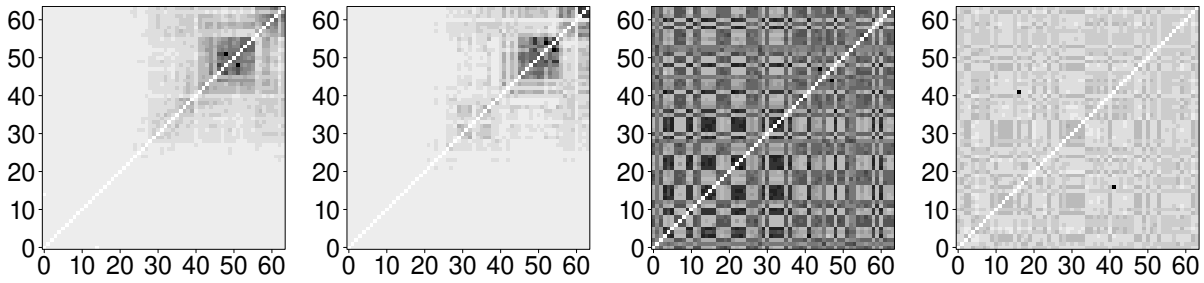
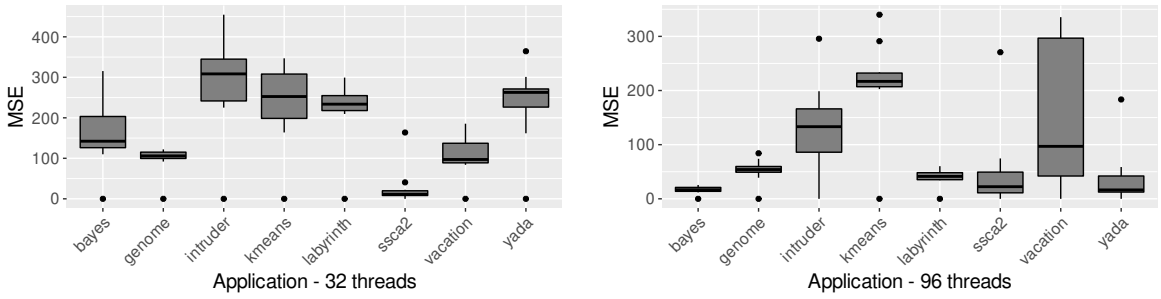


Figure 15 – Matrices with highest and lowest MSEs.

when changing the input parameters (Figure 15(a) and Figure 15(b)).

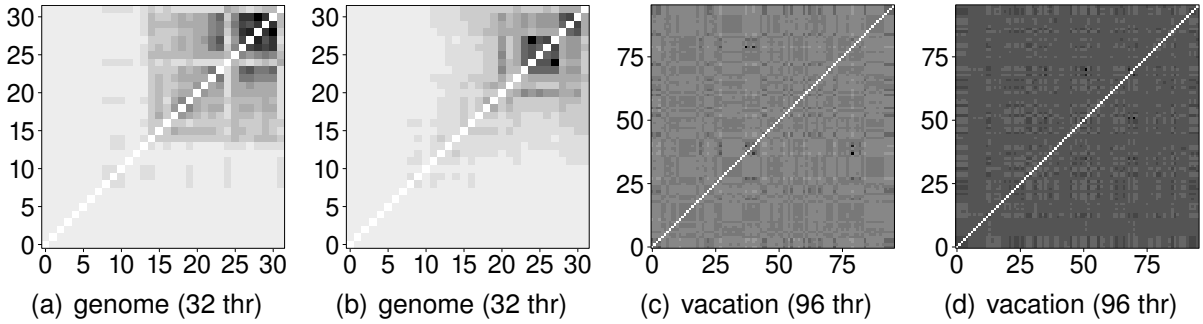
5.2.4 Stability of sharing behavior with different numbers of threads

Figure 12 in Section 5.2.2 showed the communication matrices for 64 threads. We used the same methodology to collect them for 32 and 96 threads, and show the results in Figure 16. The most different behavior occurs with *vacation* and 96 threads. However, looking at the communication pattern of two executions with the highest MSE (Figure 17(c) and Figure 17(d)) we saw the same behavior for *ssca2* in Section 5.2.2. With 96 threads, *vacation* has an all-to-all communication pattern, and the main difference between executions is the total amount of communication, which can be explained by the difference of aborts between each execution. On the other hand, applications such as *genome* present a similar behavior even changing the number of threads, for instance, with 32 threads (Figure 17(a) and Figure 17(b)).



(a) 32 threads. (b) 96 threads.

Figure 16 – Stability of the sharing behavior when changing the number of threads.



(a) genome (32 thr) (b) genome (32 thr) (c) vacation (96 thr) (d) vacation (96 thr)

Figure 17 – Matrices with the lowest and highest MSEs.

5.2.5 Dynamic behavior during execution

The goal of this experiment is to determine if the communication pattern changes during the execution of applications. For this experiment, we store multiple communication matrices in different execution phases of the applications. We analyzed the total of addresses accessed by each application (Table 4) and used it as a parameter to define a *save interval*, i.e., when to collect the communication matrix. After collection, we reset the data structure responsible to store the communication matrix. For instance, for *labyrinth* the mechanism collected eight matrices, whereas for *kmeans* nineteen matrices were collected. We run the applications using the default parameters (Table 3) and 64 threads.

In the previous sections, the MSE was compared with the first collected matrix, i.e., the baseline was the first execution. For this experiment, the baseline was the last collected matrix. For instance, after two matrices collected it is possible to calculate the MSE between them. When a third matrix is collected, we calculated the MSE between the third and the previous execution (second matrix) onward. Figure 18 presents the results.

Analyzing the graph, *ssca2* has the highest difference in communication patterns during the execution, followed by *genome*. However, as in Sections 5.2.2 and 5.2.3 the biggest difference in the communication matrices was in the amount of communication between threads since this application has an all-to-all behavior. This is visualized in

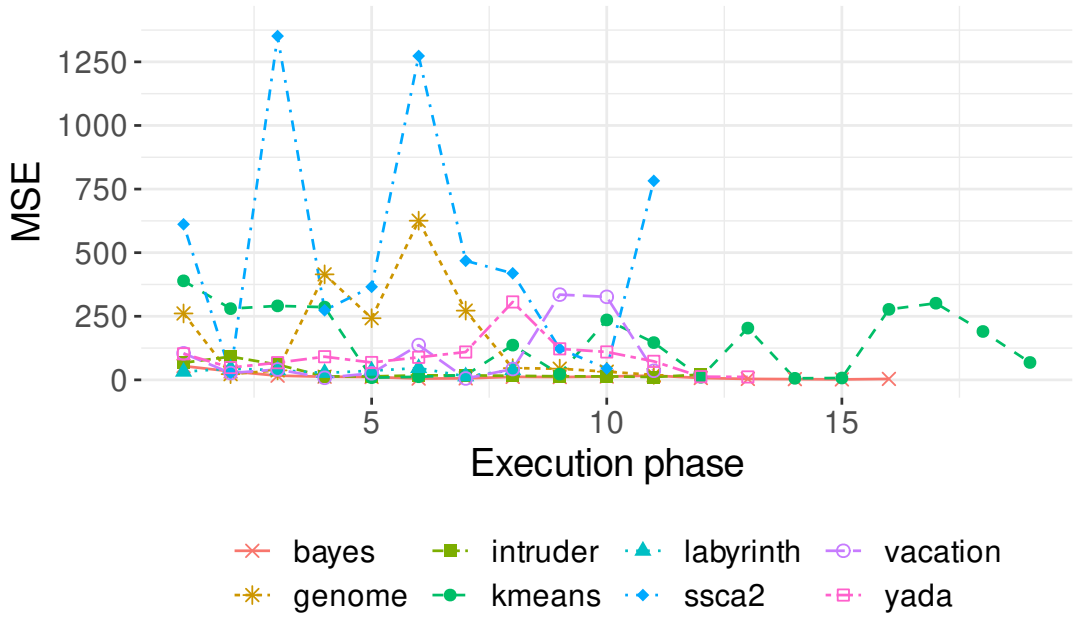


Figure 18 – Comparing the MSE on different execution phases.

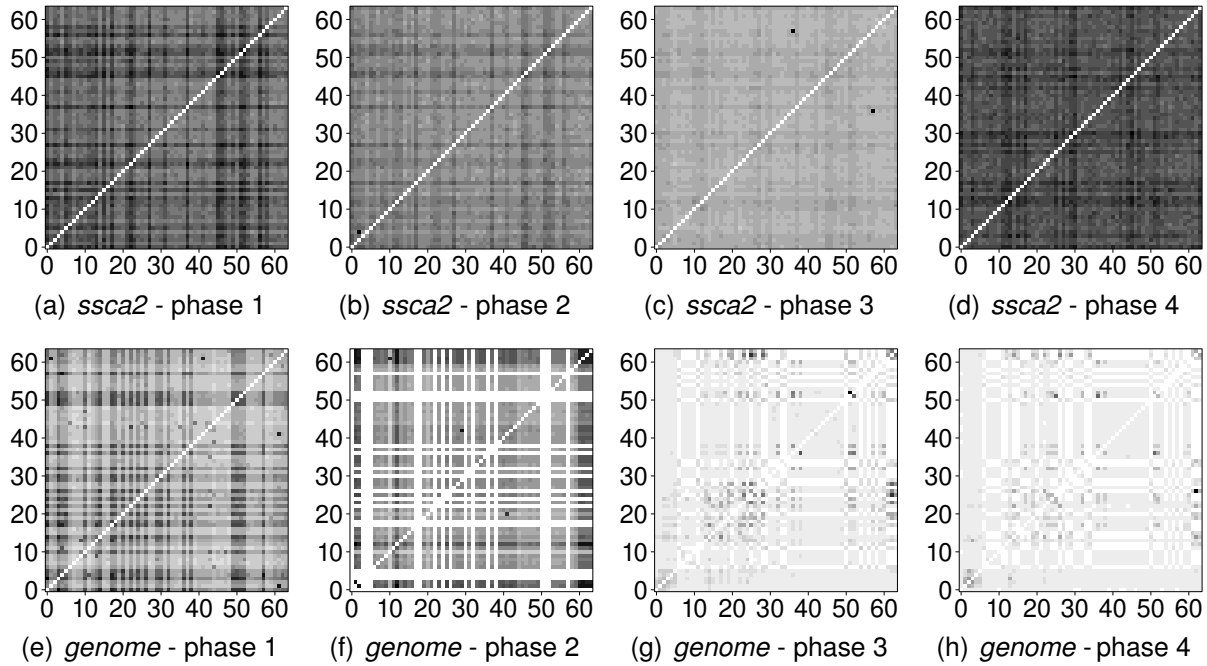


Figure 19 – Communication matrices in different execution phases.

Figure 19(a)-(d). On the other hand, *genome* has a varying communication pattern during its execution. There is an intense communication between threads in the beginning of the application, whereas in the end there is little communication (Figure 19(e)-(h)). Other applications have a similar communication behavior during their execution.

Table 6 – Kmeans performance gains with source code changes to reduce false sharing.

	Execution time		
# Threads	Baseline	Reduced false sharing	Performance gains
32	17.36s	16.01s	7.81%
64	18.05s	17.15s	4.97%
96	19.36s	17.24s	10.97%

5.3 False sharing in kmeans

We also performed an experiment to determine if the STM performance can be improved by reducing false sharing (Section 5.2.1). We selected `kmeans` for this experiment since it had the highest percentage of false sharing (Table 4). The information collected by our mechanism showed that most of false sharing happened in a matrix of floats (`new_centers`) used by `kmeans`. Analyzing its source code, we identified that the rows of `new_centers` were not padded correctly to different cache lines, take into consideration current architectures, as showed in the highlighted line (252) in Figure 20. The cache line size in current microarchitectures is typically 64 bytes. Probably the

```
246     /*
247      * Need to initialize new_centers_len and new_centers[0] to all 0.
248      * Allocate clusters on different cache lines to reduce false sharing.
249      */
250     {
251         int cluster_size = sizeof(int) + sizeof(float) * nfeatures;
252         const int cacheLineSize = 32;
253         cluster_size += (cacheLineSize-1) - ((cluster_size-1) % cacheLineSize);
254         alloc_memory = calloc(nclusters, cluster_size);
255         new_centers_len = (int**) malloc(nclusters * sizeof(int*));
256         new_centers = (float**) malloc(nclusters * sizeof(float*));
257         assert(alloc_memory && new_centers && new_centers_len);
258         for (i = 0; i < nclusters; i++) {
259             new_centers_len[i] = (int*)((char*)alloc_memory + cluster_size * i);
260             new_centers[i] = (float*)((char*)alloc_memory + cluster_size * i + sizeof(int));
261         }
262     }
```

Figure 20 – Original source code of kmeans application.

developers of the application have used a machine with a cache line size of 32 bytes. Besides, they kept this parameter fixed in the source code. We modified the source code, changing the value of the highlighted line to 64 and obtained performance gains between 4.97% and 10.97% compared to the Linux baseline, as shown in Table 6.

5.4 Summary

This chapter presented a characterization of the STAMP applications regarding their sharing behavior using the mechanism proposed in Chapter 4. Since the STAMP benchmark was developed to represent realistic workload characteristics and it covers a wide range of transactional behavior, we expect that the characterization done in this chapter could represent a wide range of real STM applications. The first experiments showed that the sharing behavior of applications does not change between executions using the same configurations, for instance, same input parameters or thread number. Besides, even during execution, the majority of applications do not present a dynamic sharing behavior. Hence, our hypothesis is that a *static* thread mapping approach is sufficient to improve the performance of the applications that are suitable for a thread mapping based on their sharing behavior. This hypothesis will be tested in Chapter (6). Beyond that, we were also able to analyze and reduce false sharing in STM memory areas, achieving performance gains when the false sharing was reduced.

6 SHARING-AWARE THREAD MAPPING IN STM

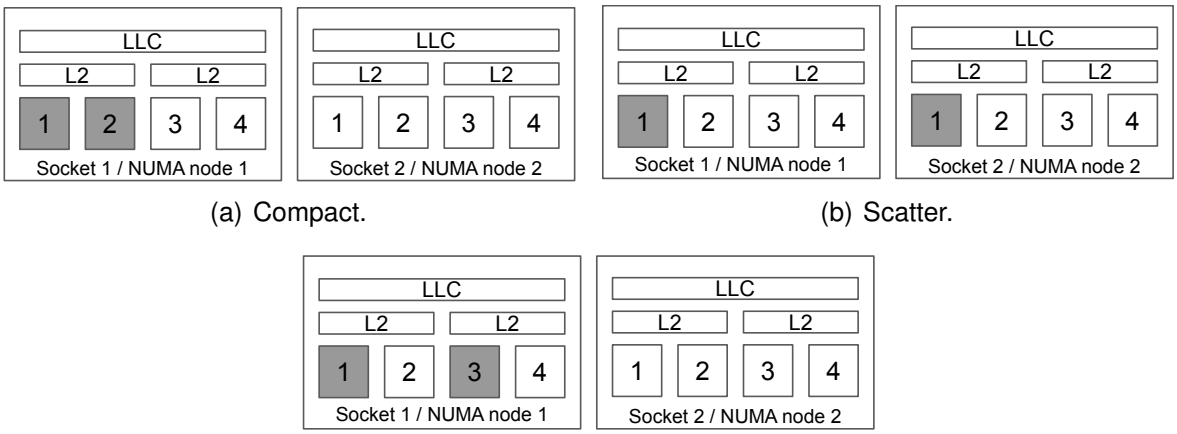
This chapter describes two mechanisms to perform sharing-aware thread mapping in STM applications. We start proposing a *static* mechanism (Section 6.1), i.e., where threads are mapped to cores at the beginning of execution, based on a previous analysis of the sharing behavior of the application. Next, an *online* mechanism is presented (Section 6.2). In an online strategy, the detection of sharing behavior and thread migration is performed based on information gathered during execution.

6.1 Static thread mapping

Since the experiments in Chapter 5 showed that `STAMP` applications do not present dynamic sharing behavior, our hypothesis is that a *static* thread mapping mechanism (where threads are mapped to cores at the beginning of execution, and never migrated) is sufficient to improve the performance of `STAMP` applications. Many tools are available to calculate a thread mapping based on a communication matrix, for instance, `Scotch` (PELLEGRINI, 1994), `TreeMatch` (JEANNOT; MERCIER; TESSIER, 2014), `EagerMap` (CRUZ et al., 2019) and `ChoiceMap` (SOOMRO; SASONGKO; UNAT, 2018). We opted to use `TopoMatch` (JEANNOT, 2020) which integrates `Scotch` and `TreeMatch` in the same library to deal with any kind of machine topology. `TopoMatch` uses the `hwloc` library (BROQUEDIS et al., 2010a) to compute the hierarchical topology of the machine. The thread mapping is calculated taking into consideration the machine topology. Hence, the idea is to use the mechanism described in Chapter 4 to extract the communication matrices of STM applications, `TopoMatch` to calculate the best thread placement and, re-execute applications with the calculated thread mapping.

6.1.1 Methodology

The applications used in the experiments were all eight benchmarks from the Stanford Transactional Applications for Multi-Processing (`STAMP`) (MINH et al., 2008) version 0.9.10, and two micro-benchmarks (HashMap and Redblacktree) from (DIEGUES; ROMANO; RODRIGUES, 2014). The input arguments used to run each application are



(a) Compact. (b) Scatter.

(c) Round-Robin.

Figure 21 – Thread mapping strategies.

shown in Table 3. To run the experiments, we used two NUMA machines with the following characteristics. Node distances were gathered with `numactl` (KLEEN, 2004):

- **Xeon**: 8 Intel Xeon E5-4650 processors and 488 GiB of RAM running Linux kernel 4.19.0-9. Each CPU has 12 2-HT cores, totaling 96 cores. Each CPU corresponds to a NUMA node (for a total of 8 NUMA nodes), and 12× 32 KB L1d, 12× 32 KB L1i, 12× 256 KB L2 and 30 MB L3 cache. Node distances: 50 – 65. Applications were compiled using gcc 8.3.0.
- **Opteron**: 4 AMD Opteron 6276 processors and 128 GiB of RAM running Linux kernel 4.15.0-96. Each CPU has 8 2-SMT cores, totaling 32 cores. Each CPU has 2 memory controllers (for a total of 8 NUMA nodes), and 16× 32 KB L1d, 8× 64 KB L1i, 8× 2 MB L2 and 2× 6 MB L3 caches. Node distances: 16 – 22. Applications were compiled using gcc 7.5.0.

Each application was executed 10 times using different mapping strategies:

- **Linux** is the default Linux CFS scheduler (WONG et al., 2008) used as our baseline.
- **Compact** places threads on sibling cores that share all cache levels, thus potentially reducing the data access latency if neighboring threads communicate (CASTRO; GÓES; MÉHAUT, 2014) (Figure 21(a)).
- **Scatter** distributes threads across different processors, avoiding cache sharing, thus, reducing memory contention (CASTRO; GÓES; MÉHAUT, 2014) (Figure 21(b)).
- **Round-Robin** is a mix between compact and scatter, where only the last level of cache is shared (CASTRO; GÓES; MÉHAUT, 2014) (Figure 21(c)).

- **Static-SharingAware (SSA)** is our proposed static approach. We first trace the behavior of each application in order to determine the communication pattern. Then, we calculate the new thread mapping offline using `TopoMatch` and re-execute the application with this *static* mapping, binding threads to cores using the function `pthread_setaffinity_np`. This approach has no runtime overhead, but is not able to handle changes in the application behavior (during execution or if the behavior is different between executions).

6.1.2 Results on the Xeon Machine

Figure 22 shows the execution time (in seconds) on the Xeon machine. Also, each bar shows the average and a confidence interval of 95%. The Static-SharingAware approach will be abbreviated as SSA in the discussion. Percentages of improvement are always compared to Linux, if not specified.

Bayes does not present a deterministic behavior, and it is not suitable to compare execution times as the order of commits at the beginning of an execution affects the final execution time (RUAN; LIU; SPEAR, 2014). Thus, we will not discuss the results of this application.

Genome has low contention and spends lots of time inside transactions (MINH et al., 2008). This application is not suitable for sharing-aware thread mapping. Even though threads with higher IDs have a well-defined pattern (Section 4.4.2), it was not enough to put them closer to increase the performance. In fact, SSA decreased the performance when compared to Linux. Round-robin had better performance gains in all scenarios. As observed by Castro, Góes and Méhaut (2014), transactions in *genome* access disjoint data most of time, hence the low contention. Thus, it is better to map threads in order to keep more cache available for each thread, i.e., a Scatter or Round-Robin mapping or similar.

Intruder has high contention and spends medium time inside transactions (MINH et al., 2008). SSA was the best mapping in 32 threads, achieving performance gains of 27.54%. However, for other thread numbers, and mainly for 96 treads, SSA decreased the performance.

Kmeans has low contention and spends little time inside transactions (MINH et al., 2008). For this application, SSA achieved good results in all thread configurations. The best performance gain of 58.28% appeared on 32 threads. For 96 threads the performance improvement of SSA was 17.14%, with Compact performing better (24.73%).

Labyrinth has high contention and spends lots of time inside transactions (MINH et al., 2008). Although this application has very different contention from *genome*, the results were similar. In that case, SSA and Compact decreased the performance, with Scatter and Round-robin having similar results. It is worth noting that the best mapping configurations performed similarly to Linux.

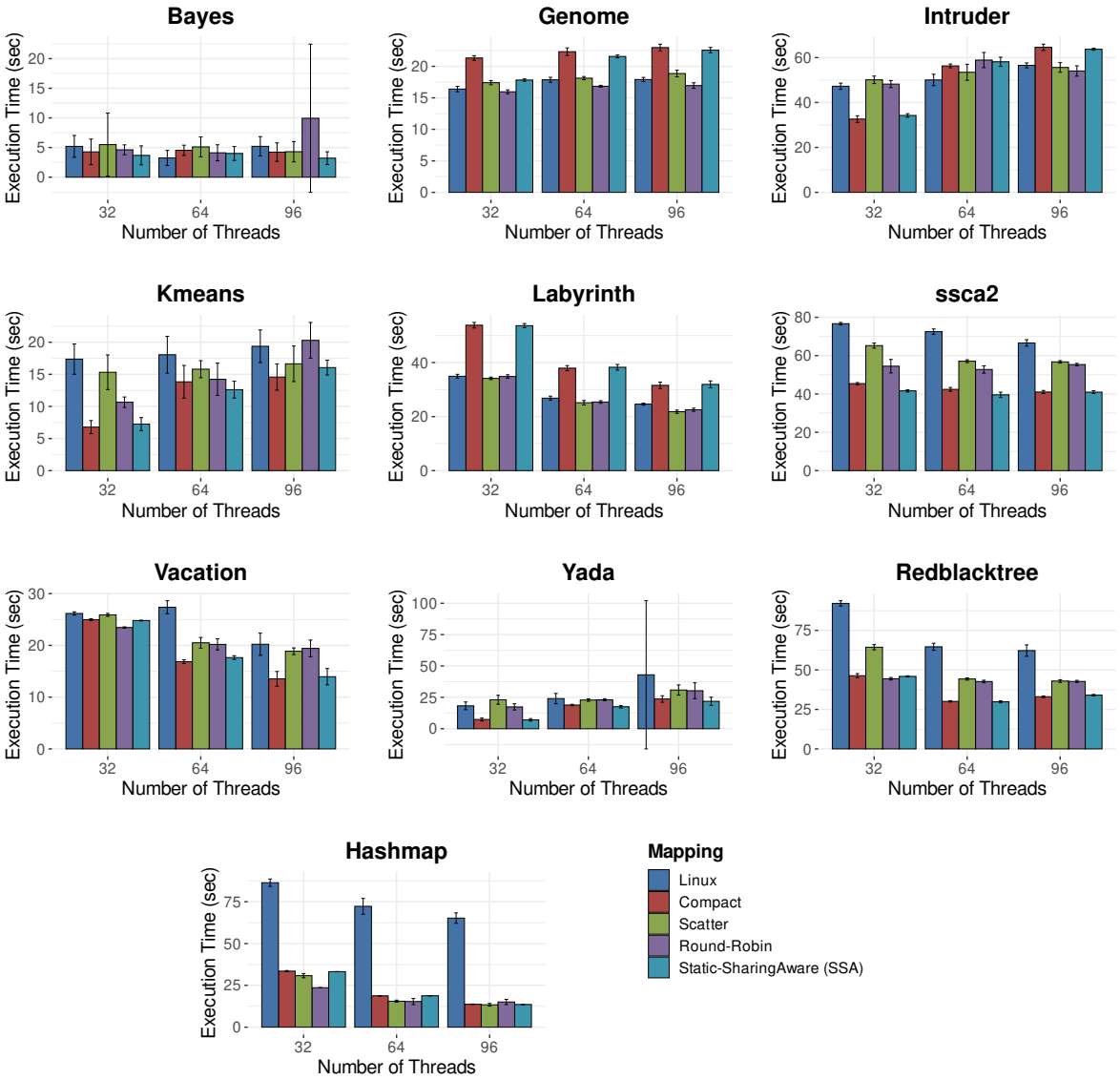


Figure 22 – Execution time results on the Xeon machine.

Ssca2 has low contention, spending little time inside transactions (MINH et al., 2008). Similar to *kmeans*, for this application SSA delivered the highest performance gain for all thread numbers used. We have a similar performance gain of 65% for 32 and 64 threads, and 38.5% for 96 threads.

Vacation has medium contention and spends high time inside transactions (MINH et al., 2008). For 32 threads, all mappings have similar results, with scatter performing slightly better. However, for 64 and 96 threads SSA had the highest performance gain of 35.55% and 31% respectively.

Yada has medium contention and spends a lot of time inside transactions (MINH et al., 2008). This is another example of an application where SSA had a good performance in all thread configurations. The highest performance gain was achieved under 32 threads, with 61.20%.

The last two micro-benchmarks, *Hashmap* and *Redblacktree* have a very similar

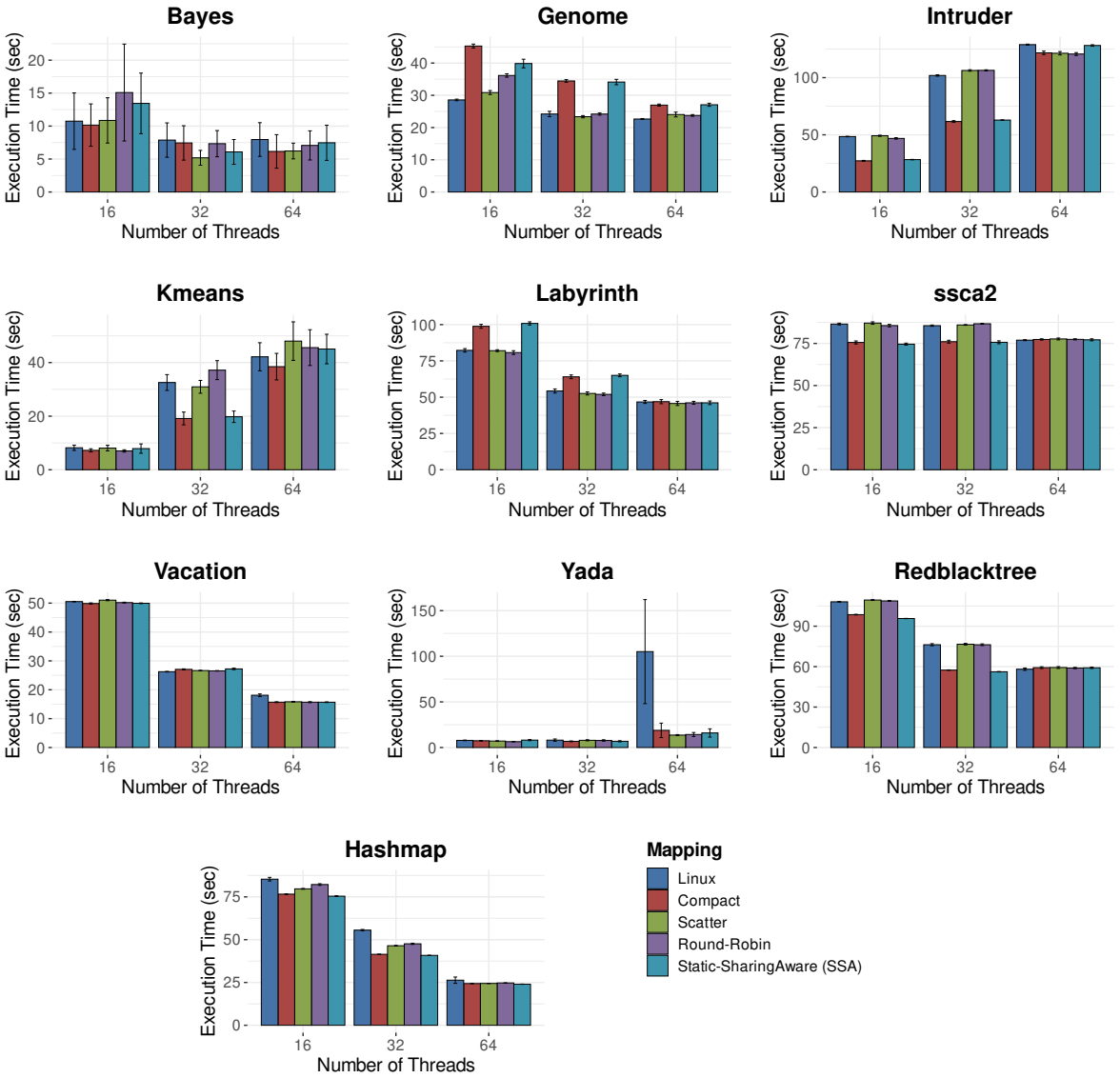


Figure 23 – Execution time results on the Opteron Machine.

communication pattern (Section 4.4.2), where communication occurs often between neighboring threads. In *Redblacktree*, SSA had the highest performance gain in 64 and 96 threads (53.9% and 45.34% respectively). In *Hashmap*, Linux had a poor performance, with all thread configurations delivering expressive performance improvements. Under 96 threads, SSA had performance gains of 79.2%.

6.1.3 Results on the Opteron Machine

Figure 23 shows the performance results on Opteron. Since the characteristics of the benchmarks were already discussed in the previous section, we will only discuss the performance results.

For *Genome*, it is possible to observe the same behavior on the Xeon machine, where SSA decreases the performance. However, for this machine Scatter was not the best mapping in all thread numbers. For 16 threads Linux performed better, and slightly

similar to Scatter and Round-Robin on 32 and 64 threads.

In the Xeon machine, *Intruder* has good performance only with 32 threads. Similarly, in Opteron, the performance gain of SSA was expressive with 32 threads (38.26%). With 16 threads, the performance improvement of SSA was even better, 41.57%. However, with 64 threads the performance started to decrease with SSA. Hence, this application is suitable for a sharing-aware thread mapping only with low thread numbers.

In *Kmeans*, SSA had an expressive performance improvement of 39.25% under 32 threads. For 16 threads, all mapping performed similarly. Under 64 threads, Compact was the best mapping.

In *Labyrinth*, it is possible to observe the same results as in the Xeon machine. This is another application that is not suitable for sharing-aware thread mapping. SSA and Compact decreased performance with 16 and 32 threads. Under 64 threads all mappings performed similarly.

For *Ssca2*, SSA delivered the highest performance gain for all thread numbers used, mainly for 16 (13.8%) and 32 threads (11.52%). Hence, the results were similar to the Xeon machine.

Vacation had a very different behavior compared to the Xeon machine. In Opteron, no mapping had a great impact on the final performance. All performed slightly similar.

Yada is similar to *Vacation*. However, in 64 threads, the Linux performance was poor, increasing execution time in more than 10$\times$.

Overall, the last two micro-benchmarks, *Hashmap* and *Redblacktree* have a very similar performance to the Xeon machine. SSA was the best mapping in both 16 and 32 threads. In *Redblacktree* the performance gains were 11.47% and 26.31% respectively, whereas in *Hashmap* it was 11.62% and 26.54%.

6.1.4 Discussion

As shown in the results, not all applications are suitable for sharing-aware thread mapping. On both machines, *Genome* and *Labyrinth* prefer a mapping that reduces memory contention, such as Scatter. However, overall, SSA improved the performance of many applications, being the mapping with the highest performance improvement. In the Xeon machine the highest performance gains of SSA over Linux were achieved by *Hashmap* using 96 threads (79.2%) and by *Yada* using 32 threads (61.2%). On the Opteron machine, the highest performance gains were achieved by *Yada* using 32 threads, achieving gains of 84.82% over Linux and by Intruder (41.57%).

It is worth noting that only transactional operations were tracked to determine the sharing behavior. The intuition is that if a memory location is shared by more than one thread, it will be protected by transactional operations.

To summarize, Table 7 shows the average performance gains of each mechanism

Table 7 – Average performance gains of each mechanism over Linux.

Machine	Compact	Scatter	RoundRobin	SSA
Xeon	15.88%	13.14%	14.50%	22.32%
Opteron	7.96%	5.86%	3.15%	6.50%

over Linux, taking into consideration all applications and thread configurations. As shown in Section 2.4 in the experiment of the array sum, the Xeon machine is more sensitive to a sharing-aware thread mapping, since `TopoMatch` prioritizes the placement of threads first inside the same socket. As explained, this machine has a larger LLC cache on each socket, showing higher gains than Opteron.

It is worth noting that the experiments so far only showed that some applications are suitable for sharing-aware thread mapping and others are not. However, at this point, we do not know which kind of characteristics make an application suitable for a sharing-aware thread mapping. In the next Section (6.2), we intend to do thread mapping during runtime. Hence, additional experiments will be made in order to discover the characteristics that make an application suitable for a sharing-aware thread mapping.

6.2 Online thread mapping

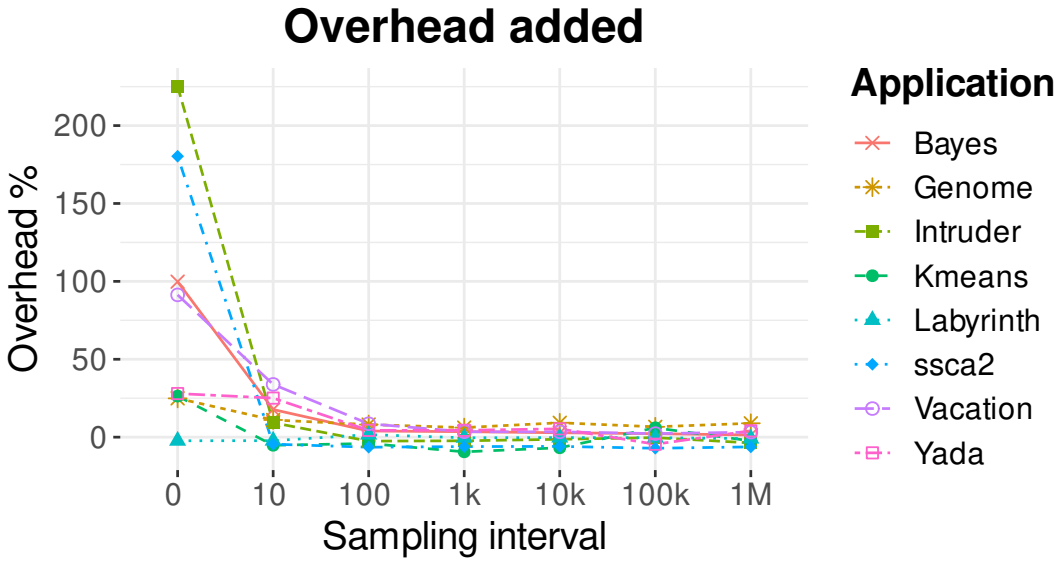
Although the experiments of the previous section (6.1) show that a static sharing-aware thread mapping is sufficient to improve the performance of STM applications, this section presents an online mechanism, called *STMap*. Contrary to SSA, STMap does not need prior information about the sharing behavior of the application, since the detection of sharing behavior and thread migration are performed based on information gathered solely during execution.

6.2.1 Reducing the overhead of online detection

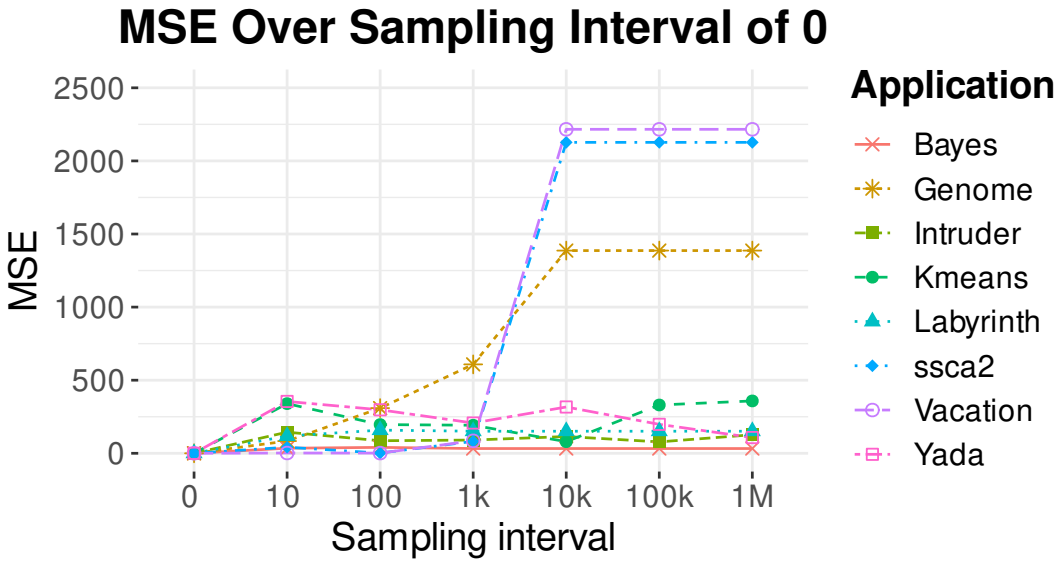
Since we are interested in detecting communication and performing thread mapping online, keeping track of every accessed address would be infeasible, due to the high overhead added to the application. Hence we use the concept of *sampling*. The goal is to choose a *sampling interval* (SI) with a high accuracy but low overhead. To choose the SI, we ran an experiment using all eight benchmarks from `STAMP`.

We start with a sampling interval of zero, i.e., all addresses are sampled in the communication matrix. The next sampling interval is 10, i.e., only execute Algorithm 2 (Chapter 4) once for every ten addresses accessed. The other sampling intervals are always ten times the previous one. To avoid contention, every thread has their own sampling interval counter.

Figure 24(a) shows the overhead results. Applications with many addresses accessed such as *Intruder* and *ssca2* have a high overhead even when using a sampling



(a) Overhead.



(b) MSE.

Figure 24 – Overhead and MSE when varying the sampling interval.

interval of 10. We also studied how the accuracy of the collected communication matrices is affected by each SI. We used the mean squared error (MSE) (Section 5.1.2) as a metric to compare the resulting matrices of each sampling interval, comparing them to the sampling interval of 0.

Figure 24(b) shows the results. Lower values are better. Analyzing the graphs, we chose a sampling interval of **100**, where the applications presented the best trade-off between overhead and accuracy.

Finally, we need to choose the *mapping interval* (MI), i.e., when the new thread mapping should be calculated. Migrating threads incurs an overhead due to cache misses and other collateral effects. In NUMA machines, these effects can be even

worse, due to messages of cache invalidation between nodes. Hence, our idea is to reduce the number of times that thread mapping is performed. We need to choose a mapping interval as early as possible, taking into consideration the trade-off between accuracy and overhead. Our mapping interval is based on the total number of accessed addresses (not just the number of sampled addresses). Thus, we made a previous analysis of the applications and chose an MI of **100,000** (more details about these thresholds will be explained in Section 6.2.2). Hence, we calculate the new thread mapping when the application accessed ‘’mapping interval” addresses. Again, to avoid contention we decided to track the total of accessed addresses of only one thread.

6.2.2 Calculating the mapping

To calculate the thread mapping, the communication matrix created by the Algorithm 2 is used as input for the `TopoMatch` mapping algorithm (JEANNOT, 2020). Using the generated mapping, threads are pinned to cores using the `pthread_setaffinity_np` function. `TopoMatch` has an important feature for NUMA machines: when calculating the new thread mapping, it tries to minimize the communication costs between sockets/nodes. Hence, it prioritizes the placement of threads first inside the same socket.

As showed in the experiments with SSA (Section 6.1), not all STM applications are suitable for sharing-aware thread mapping. Thus, we use a heuristic which is measured before calculating a new thread mapping to verify if the application would benefit from using the proposed approach. To define a heuristic, we analyzed different application characteristics. The idea is to try to discover common characteristics between applications where the sharing-aware thread mapping decreased the performance. Similar to the experiments made in Chapter 5, we use `STAMP` applications for this analysis, since it was developed to represent realistic workload characteristics and it covers a wide range of transactional behavior. To have a low overhead mechanism, we tried as much as possible to measure the desired characteristics on the main thread, avoiding thread synchronization. The initial analyzed characteristics were: the total numbers of commits and aborts, the total number of transactions, the average size of the read and write-set and the commit and abort ratios. Also, we analyzed two global metrics: the total number of accessed addresses and the amount of distinct addresses accessed. Based on Section 6.1 we already know the kinds of applications that are not suitable for sharing-aware thread mapping. Hence, we are interested in finding a heuristic to disable the mechanism on such applications. Analyzing the collected characteristics, we decided to take into consideration the number of distinct addresses (*da*) accessed by all threads and the commit and abort ratio (*cr* and *ar*) of thread 1. Besides, we used the same experiment to decide the mapping interval, i.e., the metric used to decide when a new thread mapping should be calculated. Table 8 shows the chosen metrics using

Table 8 – Characteristics used to define the heuristic and the mapping interval.

	MI: 10K, 32 Threads			MI: 50K, 32 Threads			MI: 50K, 96 Threads		
Application	*da*	*cr*	*ar*	*da*	*cr*	*ar*	*da*	*cr*	*ar*
bayes	—	—	—	—	—	—	—	—	—
genome	476	0.35	0.65	20,698	0.96	0.04	6,816	0.00	0.00
intruder	593	0.00	0.00	1,922	0.00	0.00	1,314	0.00	0.00
kmeans	19	0.06	0.94	21	0.04	0.96	20	0.00	0.00
labyrinth	4,279	1.00	0.00	14,144	0.70	0.30	—	—	—
ssca2	2,833	1.00	0.00	21,363	1.00	0.00	69,087	1.00	0.00
vacation	2,447	0.00	0.00	10,622	0.00	0.00	4,911	0.00	0.00
yada	989	0.00	0.00	1,857	0.01	0.99	7,075	0.00	0.00
redblacktree	604	0.00	0.00	2,832	0.00	0.00	10,232	0.74	0.26
hashmap	216	0.00	0.00	1,692	0.00	0.00	35,290	1.00	0.00

Algorithm 3 Heuristic used to determine if the thread mapping should be calculated

```
1:  function ENABLEMAPPING
2:      da ← getTotalDa()
3:      if (da <= da_threshold) then
4:          return true                          ▷ Compute the new thread mapping
5:      else
6:          commits ← getTotalCommitsThread()    ▷ Only for thread 1
7:          aborts ← getTotalAbortsThread()      ▷ Only for thread 1
8:          transactions ← commits + aborts
9:          if (transactions > 0) then
10:             cr ← commits/transactions
11:             ar ← aborts/transactions
12:             return ar > cr
13:         else
14:             return false   ▷ Zero transactions so far. It is not possible do calculate the ratios
```

different thread numbers and MI's. It is worth noting that in *bayes* it was not possible to collect the information, even when using a mapping interval of 10,000 addresses. This application accesses a very low number of addresses. Nevertheless, this application is not adequate to compare execution time, since its behavior is not deterministic (RUAN; LIU; SPEAR, 2014). When both *cr* and *ar* are zero in Table 8, it means that the application does not completed any transactions when the MI was reached. In that case, it was not possible to calculate the ratios. When using a mapping interval of 50,000 it was possible to define a heuristic to disable the thread mapping for the applications that were not suitable for sharing-aware thread mapping. In the case of `STAMP`, the applications less suitable for sharing-aware thread mapping are *Genome*, *Intruder* and *Labyrinth*. Since the applications that are not sensitive to our mechanism present a similar behavior on the analyzed characteristics, we expect that other applications that are not suitable for a sharing-aware thread mapping present the same characteristics. Hence, the heuristic is shown in Algorithm 3.

The `da` is calculated from the hash table (Chapter 4, Figure 5), inside the heuristic (line 2). The value of da_threshold was set to **10,000** based on the analysis of the data in Table 8. Thus, if the application had accessed less than 10,000 distinct addresses, the new thread mapping should be calculated. The intuition used here is that if there many distinct addresses accessed, the application probably does not have a well-defined communication pattern between threads. However, if there are more than 10,000 distinct addresses, the commit and abort ratio of thread 1 are used for determining if the mechanism should be disabled (lines 6 to 14). The intuition here is based on the works of Castro, Góes and Méhaut (2014) and Chan, Lam and Wang (2015), who determined that if the abort ratio is high, then the application is accessing too much shared data. Thus, putting them closer would increase cache sharing between the shared data, which is one of the main objectives of sharing-aware thread mapping. For these reasons, if the `ar` is higher than the `cr`, the new thread mapping should be calculated (line 12).

A final remark is related to the mapping interval when using 96 threads. The Xeon machine that will be used for the experiments has 96 physical cores. Analyzing the data for this configuration in Table 8, we do not see the same *da* of 10,000 for the application that we are interested in disabling the mechanism. Overall, the *da* was lower. Hence, our idea is also to include in the experiments a higher mapping interval, in that case, 100,000 addresses.

6.2.3 Final algorithm

In this Section, we present the final algorithm to detect and perform the thread mapping dynamically (Algorithm 4). Also, Figure 25 shows the updated version of the basic mechanism to detect the communication pattern, proposed in Chapter 4.

First, the thread private variable `addr_sample` (line 1) is incremented to verify if it is time to sample the memory access. Then, on the line 2 we verify if the counter of the current thread is greater than the sampling interval (Section 6.2.1). If true, we zero the variable to be able to detect the next trigger time (line 3), and Algorithm 2 is executed.

The next part of the algorithm controls when to perform the new thread mapping. In line 5, we determine if the current thread is the one responsible to control the total of addresses accessed, i.e., thread one. If true, we increment the thread private variable `total_addr` (line 6). Thus, in line 7 the algorithm determines if it is time to trigger the new thread mapping, checking if the total of accesses is greater than or equal to the mapping interval (Section 6.2.1). If true, it is necessary to verify if the application will benefit from a new mapping. This verification is done through the Algorithm 3 (line 8). Finally, if Algorithm 3 returns *true*, we compute the new thread mapping according to Section 6.2.2.

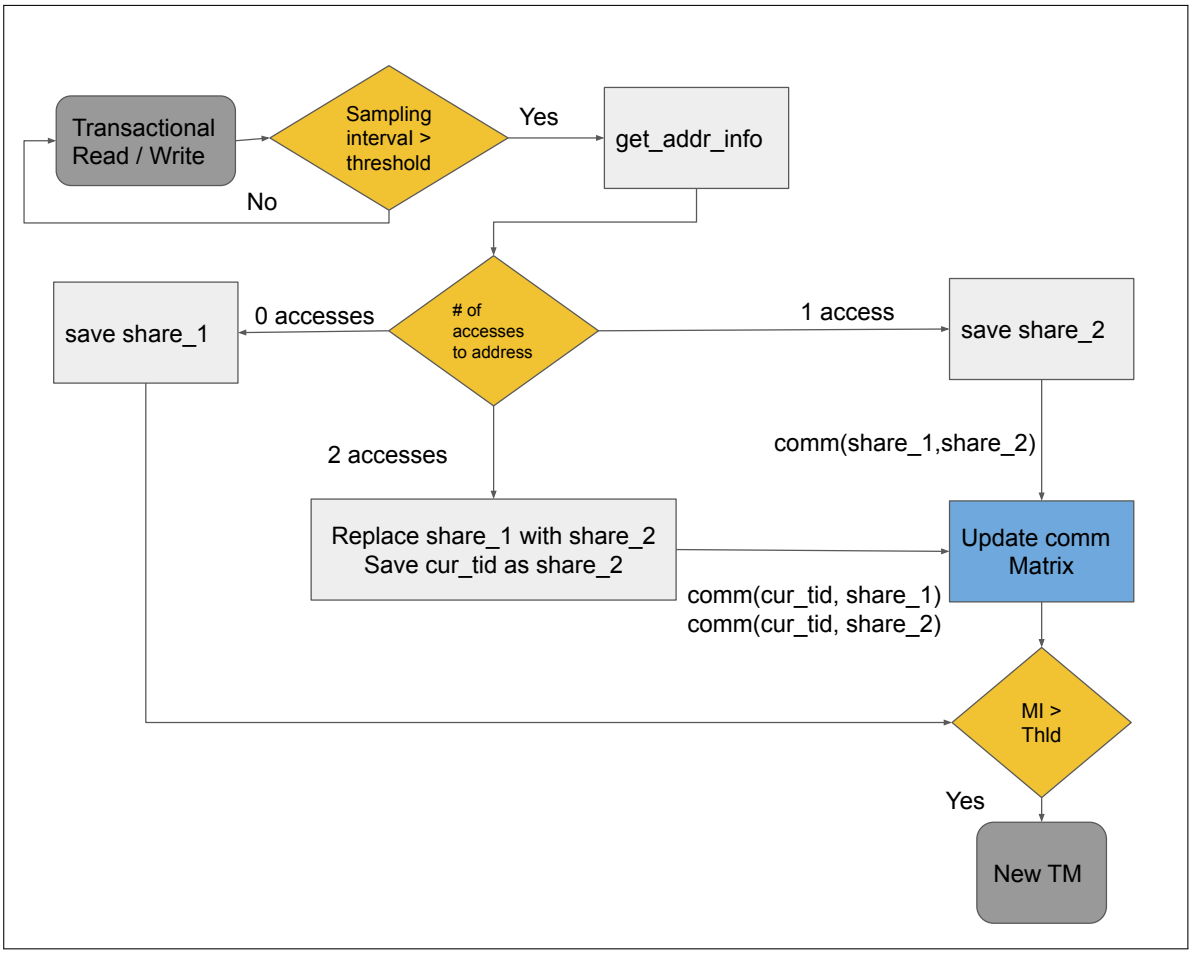


Figure 25 – Flowchart of the proposed mechanism to detect and perform thread mapping during runtime. In this Figure, Thld stands for threshold and TM for thread mapping.

Algorithm 4 Triggering communication events and thread mapping.

Require:

addr: memory address being accessed
tid: thread ID of the thread that is accessing the address
addr_sample : thread private variable used to determine if is time to sample the memory address
total_addr : thread private variable used to determine if is time to trigger the thread mapping
si: sample interval. Default 100
mi: mapping interval. Default 100,000

1: $addr\_sample \leftarrow addr\_sample + 1$
2: **if** $(addr\_sample > si)$ **then**
3: $\quad addr\_sample \leftarrow 0$
4: $\quad$ Execute algorithm 2 ▷ Proposed in Chapter 4
5: **if** $(tid = 1)$ **then**
6: $\quad total\_addr \leftarrow total\_addr + 1$
7: **if** $(tid = 1)$ **and** $(total\_addr \geq mi)$ **then**
8: $\quad$ **if** $(EnableMapping())$ **then** ▷ Algorithm 3
9: $\quad\quad$ Compute new thread mapping

6.2.4 Implementation

We extended the implementation of the proposed mechanism to detect the sharing behavior, described in Section 4.3. The extension included the Algorithm 4 inside the function `stm_write` and `stm_load`.

One key aspect of the implemented mechanism is that we only trigger the thread mapping once during the execution of the application. The experiments in Chapter 5 showed that the `STAMP` applications do not present dynamic sharing behavior, i.e., they have the same sharing pattern during all execution time. For this reason, after the first mapping interval is triggered, the mechanism is disabled, stopping to collect memory access information. Nevertheless, we do not expect significant performance penalties if the mechanism is enabled during all the execution time. As shown in Figure 24(b) the chosen sampling interval of 100 represents a very small overhead to the final execution time.

6.2.5 Results on the Xeon Machine

Figure 26 shows the execution time (in seconds) on the Xeon machine. Also, each bar shows the average and a confidence interval of 95%. When discussing the distinct addresses accessed and commit and abort ratios, these metrics are based on when the mapping interval was triggered. Percentages of improvement are always compared to Linux. As some results can be explained by the level of contention of each application, this information will be included again, for each application analyzed, such as in the static results discussion (Section 6.1.2).

Genome has little contention and spends lots of time inside transactions (MINH et al., 2008). It accesses a large number of distinct addresses, roughly twice the $da\_threshold$ metric (Section 6.2.2). Since it has little contention, i.e, the abort ratio is low, the online mechanism was disabled correctly. Although there is a large difference between SSA and STMap, Scatter performed better. Looking at 96 threads SSA had a performance loss of 26% whereas in STMap we can reduce this loss to 7.1%, by disabling the mechanism.

Intruder has high contention and spends medium time inside transactions (MINH et al., 2008). Results with 32 threads were similar for Compact and SSA. The performance gain achieved with STMap was 25% in this configuration. However, as the number of threads grows, the advantages of a sharing-aware mechanism diminish. With 64 threads the performance gains were still the best, followed by Linux, but with 96 threads we had a performance loss of 16%. This indicates that *Intruder* has a sharing pattern that is strongly related to the number of threads.

Kmeans has little contention and spends little time inside transactions (MINH et al., 2008). Due to the low contention, we can expect similar results as in *Genome*. However, this application accesses a very small amount of distinct addresses (roughly

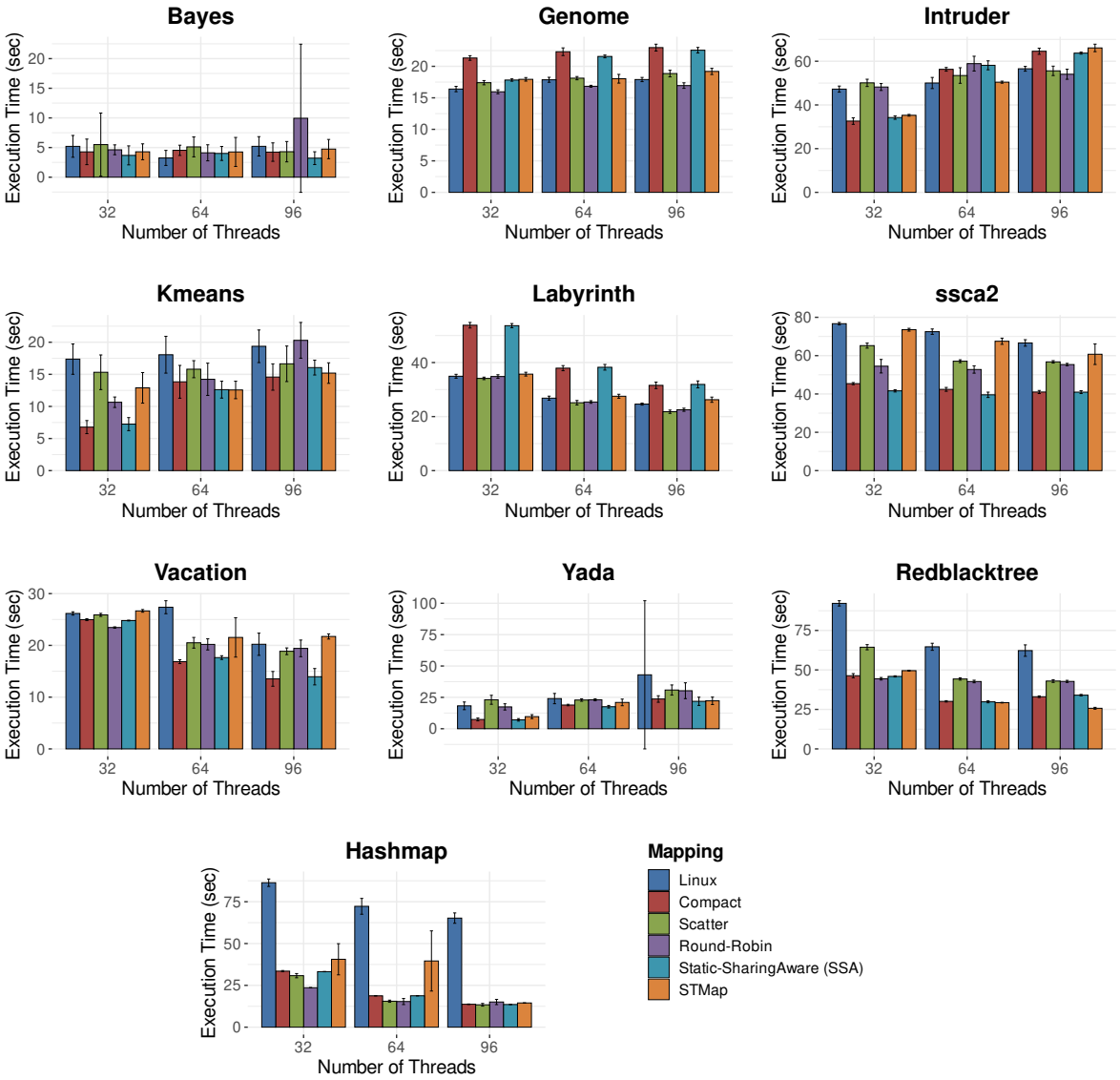


Figure 26 – Execution time results on the Xeon machine.

20), and all accesses are made to these addresses. This shows that we cannot rely only on commit and abort ratios to predict if the application is suitable for a sharing-aware thread mapping. Using 64 and 96 threads we achieved good results with STMap, with performance gains of 30.3% and 21.5%, respectively. Also, it is worth noting that with 96 threads our results were even better than the SSA mechanism. This can be explained because the application has a different sharing pattern on each execution. Thus, only an online mechanism can adapt to this behavior. In 32 threads SSA achieved performance gains of 58.3%. Hence, we expected similar results with STMap. However, we noticed an issue with the mapping interval of 100,000 used on this machine. Only in the case of 32 threads, this mapping interval was too high, and the application never triggered the mechanism.

Labyrinth has high contention and spends lots of time inside transactions (MINH et al., 2008). However, it accesses a large number of distinct addresses and the mech-

anism was correctly disabled here. Using the SSA approach we had a performance loss of 53.5% with 32 threads. However, the STMap mechanism performed similar to Linux, in all configurations. Also, it is worth noting that all mappings performed similarly, except Compact and SSA. This application shows the importance of having a heuristic to disable the mechanism, if the mechanism can predict that the final performance would be worse.

Ssca2 has little contention, spending little time inside transactions (MINH et al., 2008). The best mapping was SSA in all configurations. This application has similar characteristics as *Genome*, hence, the mechanism was disabled and the results achieved with STMap were similar to Linux.

Vacation has medium contention and spends lots of time inside transactions (MINH et al., 2008). This application has complex characteristics that are harder to predict in all thread configurations. We have three distinct cases. Using 32 threads, SSA had performance gains of 5.2% over Linux. However, due to the overhead of the mechanism, STMap was similar to Linux. Using 64 threads, the abort and commit ratio was not deterministic. We had roughly half of the times the mechanism disabled, due to a higher commit ratio. However, it is possible to see in the error bar, that sometimes the mechanism was enabled, and performed better than SSA. In 96 threads, the mechanism was incorrectly disabled all the time.

Yada has medium contention and spends lots of time inside transactions (MINH et al., 2008). It accesses a medium amount of distinct addresses. The results in all configurations of threads were similar, with Compact and SSA performing better than other mappings. STMap performed better than Linux in all thread configurations, with performance gains of 47.1%, 12.7%, and 48.2%, respectively.

Redblacktree's threads communicate often with their neighbors. Thus, Compact has a good performance, but when using 96 threads, STMap had the highest performance gain (58%) over Linux.

Hashmap has a communication pattern similar to *Redblacktree*, but STMap did not result in the highest gains using 32 and 64 threads. Specifically with 64 threads, in some runs the $da\_threshold$ was higher, and since the commit ratio of this application is high, the mechanism was disabled (Algorithm 3). The execution time varies between 19 (mechanism enabled) to 71 seconds. This explains the large error bar. However, using 96 threads the mechanism was enabled in all executions, achieving performance gains of 77.7% over Linux.

6.2.6 Results on the Opteron Machine

Figure 27 shows the performance results on Opteron. Since we already discussed the characteristics of the benchmarks in the previous section, we will only discuss the performance results.

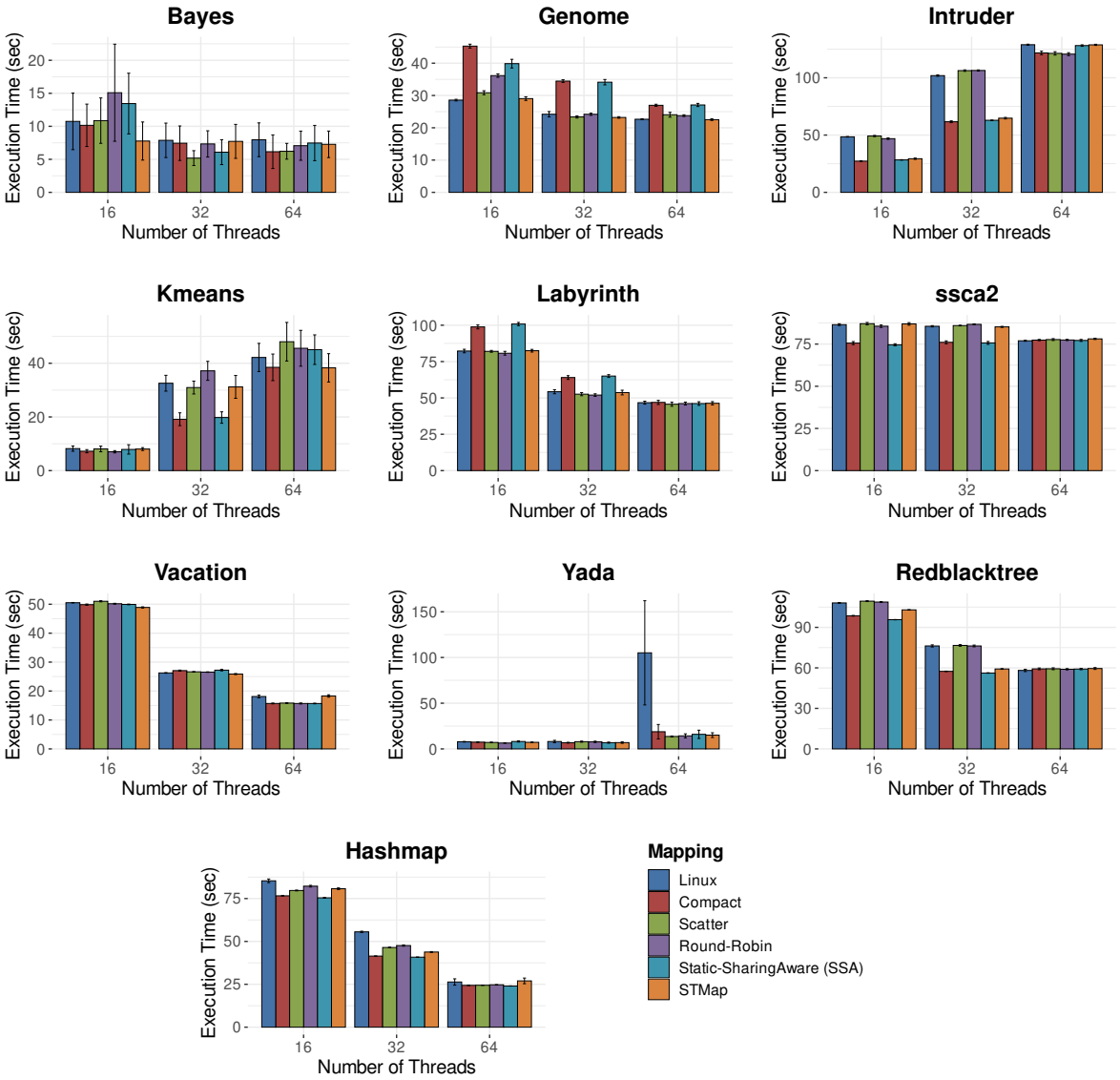


Figure 27 – Execution time results on the Opteron Machine.

Genome had a similar behavior as on the Xeon machine, and STMap was disabled correctly. Using SSA, we had a performance loss with 32 and 64 threads. With 64 threads, STMap achieved the best results, roughly equal to Linux.

In *Intruder* the results were similar for Compact and SSA, both for 32 and 64 threads. Nevertheless, the performance gain achieved with STMap with 16 threads was one of the best for the Opteron machine (39.4%). Using 32 threads, the performance gains were 36.4% compared to Linux.

In *Kmeans*, contrary to the Xeon machine, the STMap mechanism was not disabled and the performance gain achieved was up to 9,1% in 64 threads.

In *Labyrinth*, the behavior was similar to Xeon. Since it accesses a large number of distinct addresses and the commit ratio is higher, the mechanism was correctly disabled. SSA resulted in performance losses, mainly with 16 and 32 threads. On the other hand, the STMap mechanism performed similarly to Linux.

Table 9 – Average performance gains of each mechanism over Linux.

Machine	Compact	Scatter	RoundRobin	SSA	STMap
Xeon	15.88%	13.14%	14.50%	22.32%	19.07%
Opteron	7.96%	5.86%	3.15%	6.50%	9.78%

Ssca2 is another application with similar behavior on both machines. Unfortunately, since the best mapping was SSA we expected good results with STMap as well. However, as on the Xeon machine, STMap was incorrectly disabled for this application.

Vacation accesses a large number of distinct addresses and the abort ratio was slightly greater than the commit ratio in this machine. Thus, the mechanism was correctly disabled. However, no mapping had a big impact on the performance.

Yada accesses a medium number of distinct addresses (roughly 2,000). However, contrary to the Xeon machine, we did not observe big differences between the mappings, with exception of Linux with 64 threads. Nevertheless, SSA and STMap had similar results as Linux.

In the last two benchmarks, *Hashmap* and *Redblacktree*, SSA was the best mapping. STMap had slightly smaller gains compared to SSA that can be explained by the overhead in performing the mapping online.

6.2.7 Discussion

As expected, creating a unique heuristic that fits in all cases is a challenge. The main drawback of the proposed STMap mechanism appeared with *Ssca2*. On both machines, STMap was disabled by the heuristic, but we had good results using the SSA approach. On the other hand, in *Genome* and *Labyrinth* the mechanism was correctly disabled, avoiding a greater performance loss if the mechanism was enabled. Analyzing the results, sharing-aware thread mapping in STM depends on a low number of distinct addresses with a lot of accesses in these addresses. Also, applications with low and medium contention had higher gains.

Prior work on sharing-aware mapping focused solely on analyzing the communication matrix to detect if applications would benefit from a new thread mapping (DIENER et al., 2015b; BORDAGE; JEANNOT, 2018). However, our proposal to perform online sharing-aware thread mapping inside STM applications is based on the distinct addresses, commit, and abort ratios and proved to be accurate.

On the Opteron machine, the highest performance gain over Linux was achieved by *Yada* using 64 threads (85.8%) and by Intruder using 16 threads (39.4%). On the Xeon machine, the highest performance gain was achieved by *Hashmap*, achieving gains of 77.7% over Linux. To summarize, Table 9 shows the average performance gains of each mechanism over Linux, taking into consideration all applications and thread configurations.

6.2.8 Mechanism sensitivity

The STMap performance improvements are sensitivity to the mapping interval used in the heuristic to trigger the mechanism (Algorithm 4). We execute additional experiments varying the mapping interval, between 25,000 to 200,000 to verify if the chosen mapping interval of 100,000 was in fact the best one for the two machines. Figure 28 shows the performance gains achieved with each mapping interval, grouped by machine.

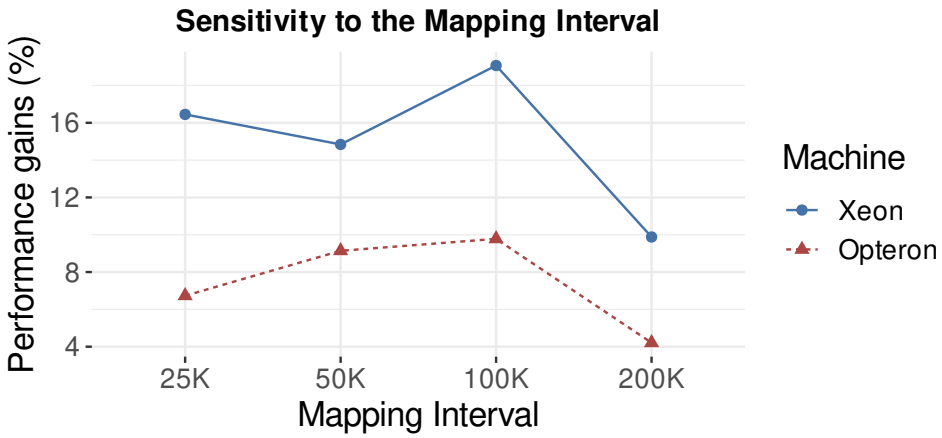


Figure 28 – Mechanism sensitivity when changing the mapping interval.

The results confirm that a mapping interval of 100,000 presents the best overall results on both machines. The mapping interval of 200,000 does not present good results because the majority of the applications do not access these quantity of addresses in one thread. Hence, for these applications the mechanism was never triggered, because the mapping interval was never reached. Although a mapping interval of 25,000 in the Xeon machine presents, on average, good results, some observations are necessary. We choose 2 applications for this discussion. Figure 29 compares the results of STMap using the best mapping interval of 100,000 to the mapping interval of 25,000.

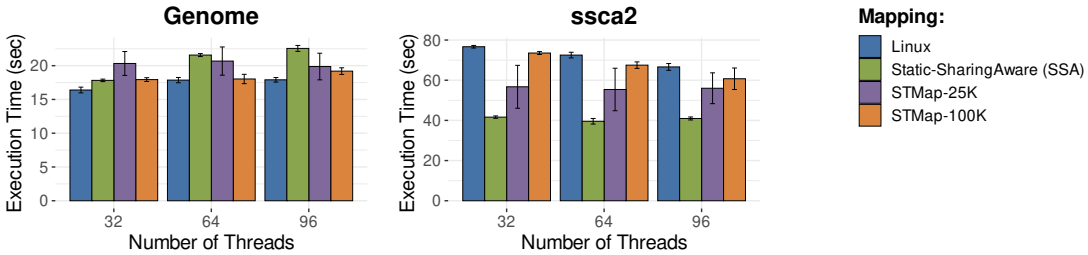


Figure 29 – Comparing STMap using different mapping intervals on the Xeon machine.

As mentioned in Section 6.2.2, *Genome* is an application that is not suitable for a sharing-aware thread mapping. Hence, it is necessary to disable the mechanism to not hurt the performance. However, as shown in Figure 29, using a mapping interval of 25,000 the mechanism was enabled. However, it was not deterministic, with sometimes

the mechanism being disabled, as shown in the large errors bars. We see similar non-determinism in *Ssca2*. As mentioned in Section 6.2.7, *Ssca2* was the main drawback of the mechanism in the mapping interval of 100,000. Nevertheless, on average, in *Ssca2* the results of the mapping interval of 25,000 were better, helping to decrease the average execution time of the mechanism using 25,000.

6.3 Summary

This chapter presented two mechanisms to perform sharing-aware thread mapping for STM applications. Although only transactional operations are tracked to extract the memory access behavior, the experimental results of the *Static-SharingAware* (SSA) mechanism shown that only the information inside the transactional memory system is enough to improve the performance of STM applications.

The next proposed mechanism, *STMap*, which is an online mechanism to extract the memory access behavior of STM applications and use this information to calculate a sharing-aware mapping of threads to cores. The main advantage is that no prior knowledge is necessary, all phases are executed while the application is running. Also, applications are not modified, only the STM system. Although we only trigger thread mapping one time, since the used applications do not present dynamic sharing behavior, we believe that the mechanism can work for a wide range of STM applications since (1) the chose sampling interval of 100 represents a very small overhead to the final execution time and (2) if the mechanism was not disabled, the next trigger time would in the mapping interval of 200,000. However, the experiments made on the mechanism sensitivity (Section 6.2.8) showed that the majority of the applications do not access these quantities of addresses in one thread.

Overall, both mechanisms showed performance gains when compared to other static thread mapping strategies and the default Linux CFS scheduler.

7 CONCLUSION

Transactional memory (TM) provides a high-level abstraction for thread synchronization in parallel programming. It can be implemented in hardware (HTM), software (STM), or both (hybrid). This thesis focused on STM. There are many studies on how to improve the performance of STM systems. Most of them focus on reducing the number of aborts, using different techniques, such as contention managements and transactional schedulers. Although reducing the number of aborts improves the performance, this thesis showed that in current multicore architectures with complex memory hierarchies it is also important to consider where the memory of the program is located and how it is accessed. Thus, the placement of threads and data is important to performance, improving the locality of memory accesses. Hence, this thesis used the concept of *sharing-aware mapping*, which aims to map threads and data of an application considering their memory access behavior. Besides, STM provides interesting mapping opportunities since the STM runtime has precise information about memory areas that are shared between threads, their respective memory addresses, and the intensity with which they are accessed by each thread.

The first contribution of this thesis (Chapter 4) is a mechanism to detect the sharing behavior of STM applications. Since the STM runtime needs the memory address on each data access operation and has precise information about shared variables, it is possible to determine the communication behavior by tracking transactional reads and writes instead of all memory accesses. Using the proposed mechanism it was possible to extract the sharing behavior of STM applications with lower overhead than other memory trace tools.

The second thesis contribution (Chapter 5) is a characterization of the sharing behavior of STM applications. This characterization is important to guide decisions regarding mapping, such as determining if an application is suitable for a thread mapping based on communication behavior and defining the type of mapping policy (static or dynamic). The main findings are that most of the characterized STM applications are suitable for a static thread mapping approach to improve the performance since (1) the applications do not present dynamic behavior and (2) the sharing pattern does not change between

executions. Furthermore, it was shown that the sharing information gathered from the STM runtime can be used to analyze and reduce false sharing in STM applications.

Using the proposed mechanism to detect the sharing behavior and the characterization of the sharing behavior of STM applications, it was proposed two mechanisms to perform sharing-aware thread mapping for STM applications. The first mechanism is *static* (Chapter 6, Section 6.1), i.e., where threads are mapped to cores at the beginning of execution, based on a previous analysis of the sharing behavior of the application. This mechanism was called as Static-SharingAware (SSA). The second mechanism, STMap (Chapter 6, Section 6.2), does not need prior information about the sharing behavior of the application, since the detection of sharing behavior and thread migration are performed based on information gathered solely during execution. In experiments with the `STAMP` benchmark suite and synthetic benchmarks, both mechanisms showed performance gains when compared to the default Linux scheduler. We conclude that applications that are suitable for a sharing-aware thread mapping, in general, present a low number of distinct addresses accessed by the STM runtime with a lot of accesses in these addresses. Also, applications with low and medium contention had higher gains. Even though the experiments were focused on the `STAMP` benchmark, we expect that the observed results can be generalized for a wide range of STM applications since `STAMP` was developed to represent realistic workload characteristics and it covers a wide range of transactional behavior.

Albeit this thesis focuses on sharing-aware thread mapping, in Appendix A we show how STMap can be extended to include sharing-aware data mapping.

7.1 Future work

The research presented in this thesis can be extended in the following ways:

- **Include balance on thread mapping**. Both SSA and STMap uses `TopoMatch` to calculated the thread mapping. As described in Section 6.2.2, when calculating the new thread mapping, `TopoMatch` tries to minimize the communication costs between sockets/nodes. Hence, it prioritizes the placement of threads first inside the same socket. However, as shown in the experiments, applications such as *Genome* and *Labyrinth* prefers mapping that reduces memory contention, i.e., balance instead of locality. This characteristic can be explored in future extensions of STMap.

- **Identify false sharing**. We showed in Chapter 5 that it is possible to identify false sharing of cache lines of STM operations using the proposed mechanism to detect the sharing behavior of STM applications. In HTM, false sharing is one of the main causes of conflicts, since the granularity of conflict detection is normally

the cache line. Hence, the mechanism proposed in this thesis could be extended to detected false sharing, reducing the number of conflicts in HTM applications.

REFERENCES

ANSARI, M. Weighted adaptive concurrency control for software transactional memory. **J. Supercomput.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 68, n. 3, p. 1027–1047, jun. 2014. ISSN 0920-8542. (Cited on page 36).

ANSARI, M.; JARVIS, K.; KOTSELIDIS, C.; LUJÁN, M.; KIRKHAM, C.; WATSON, I. Profiling transactional memory applications. In: **Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing**. Washington, DC, USA: IEEE Computer Society, 2009. (PDP '09), p. 11–20. ISBN 978-0-7695-3544-9. (Cited on page 24).

ANSARI, M.; KOTSELIDIS, C.; JARVIS, K.; LUJÁN, M.; KIRKHAM, C.; WATSON, I. Adaptive concurrency control for transactional memory. In: **MULTIPROG 2008: First Workshop on Programmability Issues for MultiCore Computers**. Springer Berlin Heidelberg, 2008. Available from Internet: <http://apt.cs.manchester.ac.uk/people/ansarim/papers/pdfs/multiprog08-ansari.pdf>. (Cited on pages 35 and 36).

ANSARI, M.; KOTSELIDIS, C.; JARVIS, K.; LUJÁN, M.; KIRKHAM, C.; WATSON, I. Advanced concurrency control for transactional memory using transaction commit rate. In: **Proceedings of the 14th International Euro-Par Conference on Parallel Processing**. Berlin, Heidelberg: Springer-Verlag, 2008. (Euro-Par '08), p. 719–728. ISBN 978-3-540-85450-0. (Cited on page 35).

ANSARI, M.; KOTSELIDIS, C.; WATSON, I.; KIRKHAM, C.; LUJÁN, M.; JARVIS, K. Lee-TM: A non-trivial benchmark suite for transactional memory. In: BOURGEOIS, A. G.; ZHENG, S. Q. (Ed.). **Algorithms and Architectures for Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 196–207. ISBN 978-3-540-69501-1. (Cited on page 28).

ANSARI, M.; LUJÁN, M.; KOTSELIDIS, C.; JARVIS, K.; KIRKHAM, C.; WATSON, I. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In: **Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers**. Berlin, Heidelberg: Springer-Verlag, 2009. (HiPEAC '09), p. 4–18. ISBN 978-3-540-92989-5. (Cited on page 37).

ANTHES, G. Researchers simplify parallel programming. **Commun. ACM**, ACM, New York, NY, USA, v. 57, n. 11, p. 13–15, oct. 2014. ISSN 0001-0782. (Cited on page 18).

ATOOFIAN, E. Speculative contention avoidance in software transactional memory. In: **2011 IEEE International Symposium on Parallel and Distributed Processing**

Workshops and PhD Forum. Washington, DC, USA: IEEE Computer Society, 2011. p. 1417–1423. ISSN 1530-2075. (Cited on page 39).

ATTIYA, H.; MILANI, A. Transactional scheduling for read-dominated workloads. In: ABDELZAHER, T.; RAYNAL, M.; SANTORO, N. (Ed.). **Principles of Distributed Systems (OPODIS 2009)**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 3–17. ISBN 9783642108778. (Cited on page 37).

ATTIYA, H.; MILANI, A. Transactional scheduling for read-dominated workloads. **Journal of Parallel and Distributed Computing**, Elsevier BV, Amsterdam, Netherlands, v. 72, n. 10, p. 1386–1396, Oct 2012. ISSN 0743-7315. (Cited on page 37).

AZIMI, R.; TAM, D. K.; SOARES, L.; STUMM, M. Enhancing operating system support for multicore processors by using hardware performance monitoring. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 43, n. 2, p. 56–65, abr. 2009. ISSN 0163-5980. (Cited on page 46).

BALDASSIN, A.; BORIN, E.; ARAUJO, G. Performance implications of dynamic memory allocators on transactional memory systems. In: **Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. New York, NY, USA: Association for Computing Machinery, 2015. (PPoPP 2015), p. 87–96. ISBN 9781450332057. (Cited on page 47).

BALDASSIN, A.; CARVALHO, J. P. L. de; GARCIA, L. A. G.; AZEVEDO, R. Energy-performance tradeoffs in software transactional memory. In: **2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing**. Washington, DC, USA: IEEE CS, 2012. p. 147–154. (Cited on page 47).

BANDEIRA, R.; DU BOIS, A. R.; PILLA, M.; VIZZOTTO, J.; MACHADO, M. Composable memory transactions for Java using a monadic intermediate language. In: PARDO, A.; SWIERSTRA, S. D. (Ed.). **Programming Languages: 19th Brazilian Symposium SBLP 2015, Belo Horizonte, Brazil, September 24-25, 2015, Proceedings**. Cham, Switzerland: Springer International Publishing, 2015. p. 128–142. ISBN 978-3-319-24012-1. (Cited on page 24).

BARRERA, I. S.; BLACK-SCHAFFER, D.; CASAS, M.; MORETÓ, M.; STUPNIKOVA, A.; POPOV, M. Modeling and optimizing numa effects and prefetching with machine learning. In: **Proceedings of the 34th ACM International Conference on Supercomputing**. New York, NY, USA: Association for Computing Machinery, 2020. (ICS '20). ISBN 9781450379830. (Cited on page 47).

BARROW-WILLIAMS, N.; FENSCH, C.; MOORE, S. A communication characterisation of Splash-2 and Parsec. In: **2009 IEEE International Symposium on Workload Characterization (IISWC)**. Washington, DC, USA: IEEE Computer Society, 2009. p. 86–97. (Cited on pages 47, 55, and 61).

BENIAMINE, D.; DIENER, M.; HUARD, G.; NAVAUX, P. O. A. TABARNAC: Visualizing and resolving memory access issues on NUMA architectures. In: **Proceedings of the 2nd Workshop on Visual Performance Analysis**. New York, NY, USA: ACM, 2015. (VPA '15), p. 1:1–1:9. ISBN 978-1-4503-4013-7. (Cited on page 44).

BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. **Commun. ACM**, ACM, New York, NY, USA, v. 13, n. 7, p. 422–426, jul. 1970. ISSN 0001-0782. (Cited on page 38).

BORDAGE, C.; JEANNOT, E. Process affinity, metrics and impact on performance: An empirical study. In: **Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing**. Piscataway, NJ, USA: IEEE Press, 2018. (CCGrid '18), p. 523–532. ISBN 978-1-5386-5815-4. (Cited on pages 29, 30, and 84).

BROQUEDIS, F.; CLET-ORTEGA, J.; MOREAUD, S.; FURMENTO, N.; GOGLIN, B.; MERCIER, G.; THIBAULT, S.; NAMYST, R. hwloc: A generic framework for managing hardware affinities in HPC applications. In: **2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing**. Washington, DC, USA: IEEE Computer Society, 2010. p. 180–186. ISSN 1066-6192. (Cited on pages 29 and 68).

BROQUEDIS, F.; FURMENTO, N.; GOGLIN, B.; WACRENIER, P.-A.; NAMYST, R. ForestGOMP: An efficient OpenMP environment for NUMA architectures. **International Journal of Parallel Programming**, Springer US, New York, NY, USA, v. 38, n. 5, p. 418–439, Oct 2010. ISSN 1573-7640. (Cited on page 45).

CALCIU, I.; SEN, S.; BALAKRISHNAN, M.; AGUILERA, M. K. How to implement any concurrent data structure for modern servers. **SIGOPS Oper. Syst. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 51, n. 1, p. 24–32, sep. 2017. ISSN 0163-5980. (Cited on page 17).

CARVALHO, J. P. L. de; HONORIO, B. C.; BALDASSIN, A.; ARAUJO, G. Improving transactional code generation via variable annotation and barrier elision. In: **2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. Washington, DC, USA: IEEE Computer Society, 2020. p. 1008–1017. (Cited on pages 28 and 29).

CASTRO, M.; GÓES, L. F. W.; FERNANDES, L. G.; MÉHAUT, J.-F. Dynamic thread mapping based on machine learning for transactional memory applications. In: KAKLAMANIS, C.; PAPATHEODOROU, T.; SPIRAKIS, P. G. (Ed.). **Euro-Par 2012 Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 465–476. ISBN 978-3-642-32820-6. (Cited on page 41).

CASTRO, M.; GÓES, L. F. W.; MÉHAUT, J.-F. Adaptive thread mapping strategies for transactional memory applications. **Journal of Parallel and Distributed Computing**, Elsevier BV, Amsterdam, Netherlands, v. 74, n. 9, p. 2845 – 2859, 2014. ISSN 0743-7315. (Cited on pages 41, 42, 48, 69, 70, and 78).

CASTRO, M.; GÓES, L. F. W.; RIBEIRO, C. P.; COLE, M.; CINTRA, M.; MÉHAUT, J. A machine learning-based approach for thread mapping on transactional memory applications. In: **2011 18th International Conference on High Performance Computing**. Washington, DC, USA: IEEE Computer Society, 2011. p. 1–10. ISSN 1094-7256. (Cited on pages 41 and 47).

CASTRO, M. B. **Improving the Performance of Transactional Memory Applications on Multicores: A Machine Learning-based Approach**. Thesis (PhD Thesis) — Université Grenoble Alpes, Grenoble, France, December 2012. (Cited on page 24).

CHAN, K.; LAM, K. T.; WANG, C.-L. Adaptive thread scheduling techniques for improving scalability of software transactional memory. In: **10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2011)**. Innsbruck, Austria: ACTAPress, 2011. p. 91–98. ISSN 9780889868649. (Cited on pages 35 and 37).

CHAN, K.; LAM, K. T.; WANG, C. L. Cache affinity optimization techniques for scaling software transactional memory systems on multi-CMP architectures. In: **2015 14th International Symposium on Parallel and Distributed Computing**. Washington, DC, USA: IEEE Computer Society, 2015. p. 56–65. ISSN 2379-5352. (Cited on pages 41, 48, and 78).

CHEN, D. D.; GIBBONS, P. B.; MOWRY, T. C. TardisTM: Incremental repair for transactional memory. In: **Proceedings of the Eleventh International Workshop on Programming Models and Applications for Multicores and Manycores**. New York, NY, USA: Association for Computing Machinery, 2020. (PMAM '20). ISBN 9781450375221. (Cited on pages 28 and 29).

CRUZ, E. H. M.; DIENER, M.; NAVAUX, P. O. A. **Thread and Data Mapping for Multicore Systems**. Cham, Switzerland: Springer International Publishing, 2018. (Cited on pages 19, 29, 42, and 45).

CRUZ, E. H. M.; DIENER, M.; PILLA, L. L.; NAVAUX, P. O. A. EagerMap: A task mapping algorithm to improve communication and load balancing in clusters of multicore systems. **ACM Trans. Parallel Comput.**, Association for Computing Machinery, New York, NY, USA, v. 5, n. 4, mar. 2019. ISSN 2329-4949. (Cited on page 68).

CRUZ, E. H. M. da; ALVES, M. A. Z.; CARISSIMI, A.; NAVAUX, P. O. A.; RIBEIRO, C. P.; MEHAUT, J. Using memory access traces to map threads and data on hierarchical multi-core platforms. In: **2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum**. Washington, DC, USA: IEEE Computer Society, 2011. p. 551–558. ISSN 1530-2075. (Cited on page 43).

DALESSANDRO, L.; SPEAR, M. F.; SCOTT, M. L. NOrec: Streamlining STM by abolishing ownership records. In: **Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. New York, NY, USA: ACM, 2010. (PPoPP '10), p. 67–78. ISBN 978-1-60558-877-3. (Cited on page 27).

DAMRON, P.; FEDOROVA, A.; LEV, Y.; LUCHANGCO, V.; MOIR, M.; NUSSBAUM, D. Hybrid transactional memory. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 41, n. 11, p. 336–346, oct. 2006. ISSN 0362-1340. (Cited on page 22).

DASHTI, M.; FEDOROVA, A.; FUNSTON, J.; GAUD, F.; LACHAIZE, R.; LEPERS, B.; QUEMA, V.; ROTH, M. Traffic management: A holistic approach to memory placement on NUMA systems. In: **Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems**. New York, NY, USA: ACM, 2013. (ASPLOS '13), p. 381–394. ISBN 978-1-4503-1870-9. (Cited on page 46).

DAVID, T.; GUERRAOUI, R.; TRIGONAKIS, V. Everything you always wanted to know about synchronization but were afraid to ask. In: **Proceedings of the Twenty-Fourth**

ACM Symposium on Operating Systems Principles. New York, NY, USA: ACM, 2013. (SOSP '13), p. 33–48. ISBN 978-1-4503-2388-8. (Cited on page 17).

DENOYELLE, N.; GOGLIN, B.; JEANNOT, E.; ROPARS, T. Data and thread placement in NUMA architectures: A statistical learning approach. In: **Proceedings of the 48th International Conference on Parallel Processing**. New York, NY, USA: ACM, 2019. (ICPP 2019), p. 39:1–39:10. ISBN 978-1-4503-6295-5. (Cited on pages 44 and 47).

DI SANZO, P. Analysis, classification and comparison of scheduling techniques for software transactional memories. **IEEE Transactions on Parallel and Distributed Systems**, Institute of Electrical and Electronics Engineers (IEEE), Los Alamitos, CA, USA, v. 28, n. 12, p. 3356–3373, dec 2017. (Cited on pages 19, 34, 35, 37, and 38).

DI SANZO, P.; PELLEGRINI, A.; SANNICANDRO, M.; CICIANI, B.; QUAGLIA, F. Adaptive model-based scheduling in software transactional memory. **IEEE Transactions on Computers**, Institute of Electrical and Electronics Engineers (IEEE), Washington, DC, USA, v. 69, n. 5, p. 621–632, May 2020. ISSN 2326-3814. (Cited on pages 28, 29, and 41).

DI SANZO, P.; RE, F. D.; RUGHETTI, D.; CICIANI, B.; QUAGLIA, F. Regulating concurrency in software transactional memory: An effective model-based approach. In: **2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems**. Washington, DC, USA: IEEE Computer Society, 2013. p. 31–40. ISSN 1949-3673. (Cited on page 40).

DI SANZO, P.; SANNICANDRO, M.; CICIANI, B.; QUAGLIA, F. Markov chain-based adaptive scheduling in software transactional memory. In: **2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. Washington, DC, USA: IEEE Computer Society, 2016. (Cited on page 41).

DICE, D.; SHALEV, O.; SHAVIT, N. Transactional Locking II. In: **Proceedings of the 20th International Conference on Distributed Computing**. Berlin, Heidelberg: Springer-Verlag, 2006. (DISC'06), p. 194–208. ISBN 3-540-44624-9, 978-3-540-44624-8. (Cited on pages 18, 25, and 26).

DICE, D.; SHAVIT, N. Understanding tradeoffs in software transactional memory. In: **Proceedings of the International Symposium on Code Generation and Optimization**. Washington, DC, USA: IEEE Computer Society, 2007. (CGO '07), p. 21–33. ISBN 0-7695-2764-7. (Cited on pages 18 and 28).

DIEGUES, N.; ROMANO, P.; RODRIGUES, L. Virtues and limitations of commodity hardware transactional memory. In: **2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)**. New York, NY, USA: Association for Computing Machinery, 2014. (PACT '14), p. 3–14. ISBN 9781450328098. (Cited on pages 54 and 68).

DIENER, M.; CRUZ, E. H.; NAVAUX, P. O.; BUSSE, A.; HEIß, H.-U. Communication-aware process and thread mapping using online communication detection. **Parallel Computing**, v. 43, p. 43–63, 2015. ISSN 0167-8191. (Cited on page 46).

DIENER, M.; CRUZ, E. H.; PILLA, L. L.; DUPROS, F.; NAVAUX, P. O. Characterizing communication and page usage of parallel applications for thread and data mapping. **Performance Evaluation**, Elsevier BV, Amsterdam, Netherlands, v. 88-89, p. 18–36, jun 2015. (Cited on pages 44, 47, 50, 55, and 84).

DIENER, M.; CRUZ, E. H. M.; ALVES, M. A. Z.; NAVAUX, P. O. A.; KOREN, I. Affinity-based thread and data mapping in shared memory systems. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 49, n. 4, p. 64:1–64:38, dec. 2016. ISSN 0360-0300. (Cited on pages 29, 42, 43, and 45).

DIENER, M.; CRUZ, E. H. M.; ALVES, M. A. Z.; NAVAUX, P. O. A. Communication in shared memory: Concepts, definitions, and efficient detection. In: **2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)**. Washington, DC, USA: IEEE Computer Society, 2016. p. 151–158. ISSN 2377-5750. (Cited on pages 30, 51, and 59).

DIENER, M.; CRUZ, E. H. M.; ALVES, M. A. Z.; NAVAUX, P. O. A.; BUSSE, A.; HEISS, H. U. Kernel-based thread and data mapping for improved memory affinity. **IEEE Transactions on Parallel and Distributed Systems**, v. 27, n. 9, p. 2653–2666, Sept 2016. ISSN 1045-9219. (Cited on page 46).

DIENER, M.; MADRUGA, F. L.; RODRIGUES, E. L.; ALVES, M. A. Z.; SCHNEIDER, J.; NAVAUX, P. O. A.; HEISS, H.-U. Evaluating thread placement based on memory access patterns for multi-core processors. In: **2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)**. Washington, DC, USA: IEEE Computer Society, 2010. p. 491–496. (Cited on page 43).

DOLEV, S.; HENDLER, D.; SUISSA, A. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In: **Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 2008. (PODC '08), p. 125–134. ISBN 978-1-59593-989-0. (Cited on page 37).

DRAGOJEVIĆ, A.; FELBER, P.; GRAMOLI, V.; GUERRAOUI, R. Why STM can be more than a research toy. **Commun. ACM**, ACM, New York, NY, USA, v. 54, n. 4, p. 70–77, abr. 2011. ISSN 0001-0782. (Cited on page 18).

DRAGOJEVIĆ, A.; GUERRAOUI, R.; KAPALKA, M. Stretching transactional memory. In: **Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2009. (PLDI '09), p. 155–165. ISBN 978-1-60558-392-1. (Cited on page 25).

DRAGOJEVIĆ, A.; GUERRAOUI, R.; SINGH, A. V.; SINGH, V. Preventing versus curing: Avoiding conflicts in transactional memories. In: **Proceedings of the 28th ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 2009. (PODC '09), p. 7–16. ISBN 978-1-60558-396-9. (Cited on pages 34 and 38).

DUPROS, F.; RIBEIRO, C. P.; CARISSIMI, A.; MÉHAUT, J.-F. Parallel simulations of seismic wave propagation on NUMA architectures. In: CHAPMAN, B.; DESPREZ, F.; JOUBERT, G. R.; LICHNEWSKY, A.; PETERS, F.; PRIOL, T. (Ed.). **Parallel Computing: From Multicores and GPU's to Petascale**. Amsterdam, Netherlands: IOS Press,

2010, (Advances in Parallel Computing, v. 19). p. 67–74. ISBN 978-1-60750-529-7. (Cited on page 43).

ENNALS, R. **Software Transactional Memory Should Not Be Obstruction-Free**. Cambridge, UK, 2006. (Cited on page 25).

ERANIAN, S. **The perfmon2 interface specification**. Palo Alto, CA, USA, 2005. Available from Internet: <https://www.hpl.hp.com/techreports/2004/HPL-2004-200R1.pdf>. (Cited on page 45).

FELBER, P.; FETZER, C.; RIEGEL, T.; MARLIER, P. Time-based software transactional memory. **IEEE Transactions on Parallel & Distributed Systems**, IEEE Computer Society, Los Alamitos, CA, USA, v. 21, p. 1793–1807, 2010. ISSN 1045-9219. (Cited on pages 10, 23, 25, 26, 31, and 53).

FRASER, K.; HARRIS, T. Concurrent programming without locks. **ACM Trans. Comput. Syst.**, ACM, New York, NY, USA, v. 25, n. 2, may 2007. ISSN 0734-2071. (Cited on page 18).

GAUD, F.; LEPERS, B.; FUNSTON, J.; DASHTI, M.; FEDOROVA, A.; QUÉMA, V.; LACHAIZE, R.; ROTH, M. Challenges of memory management on modern NUMA systems. **Commun. ACM**, ACM, New York, NY, USA, v. 58, n. 12, p. 59–66, nov. 2015. ISSN 0001-0782. (Cited on pages 17 and 29).

GÓES, L. F.; RIBEIRO, C. P.; CASTRO, M.; MÉHAUT, J.-F.; COLE, M.; CINTRA, M. Automatic skeleton-driven memory affinity for transactional worklist applications. **Int. J. Parallel Program.**, Kluwer Academic Publishers, USA, v. 42, n. 2, p. 365–382, abr. 2014. ISSN 0885-7458. (Cited on pages 42 and 48).

GRAHN, H. Transactional memory. **J. Parallel Distrib. Comput.**, Academic Press, Inc., Orlando, FL, USA, v. 70, n. 10, p. 993–1008, oct. 2010. ISSN 0743-7315. (Cited on pages 17, 18, 22, 23, and 24).

GRAMOLI, V.; GUERRAOUI, R. Democratizing transactional programming. **Commun. ACM**, ACM, New York, NY, USA, v. 57, n. 1, p. 86–93, jan. 2014. ISSN 0001-0782. (Cited on page 18).

GRAMOLI, V.; GUERRAOUI, R. Programming with Transactional Memory. In: PLLANA, S.; XHAFA, F. (Ed.). **Programming multi-core and many-core computing systems**. Hoboken, New Jersey, EUA: John Wiley & Sons, Ltd, 2017. chp. 8, p. 165–183. ISBN 9781119332015. (Cited on page 18).

GUERRAOUI, R.; HERLIHY, M.; POCHON, B. Towards a theory of transactional contention managers. In: **Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 2006. (PODC '06), p. 316–317. ISBN 1-59593-384-0. (Cited on page 24).

GUERRAOUI, R.; KAPALKA, M. On the correctness of transactional memory. In: **Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. New York, NY, USA: ACM, 2008. (PPoPP '08), p. 175–184. ISBN 978-1-59593-795-7. (Cited on page 23).

GUERRAOUI, R.; KAPALKA, M.; VITEK, J. STMBench7: A benchmark for software transactional memory. In: **Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007**. New York, NY, USA: ACM, 2007. (EuroSys '07), p. 315–324. ISBN 978-1-59593-636-3. (Cited on page 28).

HARRIS, T.; FRASER, K. Language support for lightweight transactions. In: **Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications**. New York, NY, USA: Association for Computing Machinery, 2003. (OOPSLA '03), p. 388–402. ISBN 1581137125. (Cited on page 28).

HARRIS, T.; LARUS, J.; RAJWAR, R. **Transactional Memory, 2nd Edition**. 2nd. ed. San Rafael, California, USA: Morgan and Claypool Publishers, 2010. ISBN 1608452352, 9781608452354. (Cited on pages 18, 22, 23, 24, 25, and 51).

HEBER, T.; HENDLER, D.; SUISSA, A. On the impact of serializing contention management on stm performance. In: ABDELZAHER, T.; RAYNAL, M.; SANTORO, N. (Ed.). **Principles of Distributed Systems (OPODIS 2009)**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 225–239. ISBN 978-3-642-10877-8. (Cited on page 39).

HEBER, T.; HENDLER, D.; SUISSA, A. On the impact of serializing contention management on stm performance. **Journal of Parallel and Distributed Computing**, Berlin, Heidelberg, v. 72, n. 6, p. 739 – 750, 2012. ISSN 0743-7315. (Cited on page 39).

HENDLER, D.; SUISSA-PELEG, A. Scheduling-based contention management techniques for transactional memory. In: GUERRAOUI, R.; ROMANO, P. (Ed.). **Transactional Memory. Foundations, Algorithms, Tools, and Applications: COST Action Euro-TM IC1001**. Cham, Switzerland: Springer International Publishing, 2015. p. 213–227. ISBN 978-3-319-14720-8. (Cited on page 34).

HERLIHY, M.; LUCHANGCO, V.; MOIR, M.; SCHERER III, W. N. Software transactional memory for dynamic-sized data structures. In: **Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 2003. (PODC '03), p. 92–101. ISBN 1-58113-708-7. (Cited on pages 25 and 28).

HERLIHY, M.; MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In: **Proceedings of the 20th Annual International Symposium on Computer Architecture**. New York, NY, USA: ACM, 1993. (ISCA '93), p. 289–300. ISBN 0-8186-3810-9. (Cited on page 22).

HERLIHY, M.; SHAVIT, N. **The Art of Multiprocessor Programming**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 0123705916, 9780123705914. (Cited on page 18).

HONG, S.; OGUNTEBI, T.; CASPER, J.; BRONSON, N.; KOZYRAKIS, C.; OLUKOTUN, K. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In: **Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)**. Washington, DC, USA: IEEE Computer Society, 2010. (IISWC '10), p. 1–11. ISBN 978-1-4244-9297-8. (Cited on page 28).

HUGHES, C.; POE, J.; QOUNEH, A.; LI, T. On the (dis)similarity of transactional memory workloads. In: **2009 IEEE International Symposium on Workload Characterization (IISWC)**. Washington, DC, USA: IEEE Computer Society, 2009. p. 108–117. (Cited on page 47).

JANERT, P. K. **Feedback Control for Computer Systems: Introducing Control Theory to Enterprise Programmers**. Sebastopol, CA, USA: O'Reilly Media, 2013. 330 pages. ISBN 1449361692. (Cited on page 35).

JEANNOT, E. **TopoMatch: Process mapping algorithms and tools for general topologies**. 2020. <https://gitlab.inria.fr/ejeannot/topomatch>. Last accessed 6th, January 2021. (Cited on pages 68 and 76).

JEANNOT, E.; MENESES, E.; MERCIER, G.; TESSIER, F.; ZHENG, G. Communication and topology-aware load balancing in Charm++ with TreeMatch. In: **2013 IEEE International Conference on Cluster Computing (CLUSTER)**. Washington, DC, USA: IEEE Computer Society, 2013. p. 1–8. (Cited on page 19).

JEANNOT, E.; MERCIER, G.; TESSIER, F. Process placement in multicore clusters: Algorithmic issues and practical techniques. **IEEE Trans. Parallel Distrib. Syst.**, IEEE Press, Washington, DC, USA, v. 25, n. 4, p. 993–1002, 2014. ISSN 1045-9219. (Cited on page 68).

JIN, H.; FRUMKIN, M.; YAN, J. **The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance**. Moffett Field, CA, 1999. Available from Internet: <https://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf>. (Cited on page 43).

KESTOR, G.; KARAKOSTAS, V.; UNSAL, O. S.; CRISTAL, A.; HUR, I.; VALERO, M. RMS-TM: A comprehensive benchmark suite for transactional memory systems. In: **Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering**. New York, NY, USA: Association for Computing Machinery, 2011. (ICPE '11), p. 335–346. ISBN 9781450305198. (Cited on page 28).

KLEEN, A. **An NUMA API for Linux**. Fürth, Germany, 2004. Available from Internet: <http://www.halobates.de/numaapi3.pdf>. (Cited on pages 43, 54, 69, 110, and 111).

KLUG, T.; OTT, M.; WEIDENDORFER, J.; TRINITIS, C. autopin – automated optimization of thread-to-core pinning on multicore systems. In: STENSTROM, P. (Ed.). **Transactions on High-Performance Embedded Architectures and Compilers III**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 219–235. ISBN 978-3-642-19448-1. (Cited on page 45).

LAMETER, C. NUMA (Non-Uniform Memory Access): An overview. **Queue**, ACM, New York, NY, USA, v. 11, n. 7, p. 40:40–40:51, jul. 2013. ISSN 1542-7730. (Cited on page 29).

LARUS, J.; KOZYRAKIS, C. Transactional memory. **Commun. ACM**, ACM, New York, NY, USA, v. 51, n. 7, p. 80–88, jul. 2008. ISSN 0001-0782. (Cited on page 24).

LEPERS, B.; QUÉMA, V.; FEDOROVA, A. Thread and memory placement on NUMA systems: Asymmetry matters. In: **Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference**. Berkeley, CA, USA: USENIX Association, 2015. (USENIX ATC '15), p. 277–289. ISBN 978-1-931971-225. (Cited on page 17).

LÖF, H.; HOLMGREN, S. Affinity-on-next-touch: Increasing the performance of an industrial PDE solver on a cc-NUMA system. In: **Proceedings of the 19th Annual International Conference on Supercomputing**. New York, NY, USA: ACM, 2005. (ICS '05), p. 387–392. ISBN 1-59593-167-8. (Cited on page 46).

LOMET, D. B. Process structuring, synchronization, and recovery using atomic actions. In: **Proceedings of an ACM Conference on Language Design for Reliable Software**. New York, NY, USA: ACM, 1977. p. 128–137. (Cited on page 22).

LUK, C.-K.; COHN, R.; MUTH, R.; PATIL, H.; KLAUSER, A.; LOWNEY, G.; WALLACE, S.; REDDI, V. J.; HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In: **Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2005. (PLDI '05), p. 190–200. ISBN 1-59593-056-6. (Cited on pages 43, 44, and 56).

MAJO, Z.; GROSS, T. R. Matching memory access patterns and data placement for NUMA systems. In: **Proceedings of the Tenth International Symposium on Code Generation and Optimization**. New York, NY, USA: ACM, 2012. (CGO '12), p. 230–241. ISBN 978-1-4503-1206-6. (Cited on page 44).

MAJO, Z.; GROSS, T. R. A library for portable and composable data locality optimizations for NUMA systems. In: **Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. New York, NY, USA: ACM, 2015. (PPoPP 2015), p. 227–238. ISBN 978-1-4503-3205-7. (Cited on page 43).

MAJO, Z.; GROSS, T. R. A library for portable and composable data locality optimizations for NUMA systems. **ACM Trans. Parallel Comput.**, ACM, New York, NY, USA, v. 3, n. 4, p. 20:1–20:32, mar. 2017. ISSN 2329-4949. (Cited on page 43).

MALDONADO, W.; MARLIER, P.; FELBER, P.; SUISSA, A.; HENDLER, D.; FEDOROVA, A.; LAWALL, J. L.; MULLER, G. Scheduling support for transactional memory contention management. In: **Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. New York, NY, USA: ACM, 2010. (PPoPP '10), p. 79–90. ISBN 978-1-60558-877-3. (Cited on page 40).

MALDONADO, W.; MARLIER, P.; FELBER, P.; LAWALL, J.; MULLER, G.; RIVIÈRE, E. Deadline-aware scheduling for software transactional memory. In: **2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)**. Washington, DC, USA: IEEE Computer Society, 2011. p. 257–268. ISSN 2158-3927. (Cited on page 40).

MARIANO, A.; DIENER, M.; BISCHOF, C.; NAVAUX, P. O. A. Analyzing and improving memory access patterns of large irregular applications on NUMA machines. In: **2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)**. Washington, DC, USA: IEEE Computer Society, 2016. p. 382–387. ISSN 2377-5750. (Cited on pages 44 and 112).

MARQUES, A.; BALDASSIN, A. Energy-aware scheduling in transactional memory systems. In: **Proceedings of the 29th Symposium on Integrated Circuits and Systems Design: Chip on the Mountains**. Piscataway, NJ, USA: IEEE Press, 2016. (SBCCI '16), p. 26:1–26:6. ISBN 978-1-5090-2736-1. (Cited on page 41).

MAZAHERI, A.; WOLF, F.; JANNESARI, A. Unveiling thread communication bottlenecks using hardware-independent metrics. In: **Proceedings of the 47th International Conference on Parallel Processing**. New York, NY, USA: ACM, 2018. (ICPP 2018). ISBN 9781450365109. (Cited on page 30).

MCVOY, L.; STAELIN, C. Lmbench: Portable tools for performance analysis. In: **Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference**. USA: USENIX Association, 1996. (ATEC '96), p. 23. (Cited on page 112).

MICCIANCIO, D.; REGEV, O. Lattice-based cryptography. In: BERNSTEIN, D. J.; BUCHMANN, J.; DAHMEN, E. (Ed.). **Post-Quantum Cryptography**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 147–191. ISBN 978-3-540-88702-7. (Cited on page 44).

MINH, C. C.; CHUNG, J.; KOZYRAKIS, C.; OLUKOTUN, K. STAMP: Stanford Transactional Applications for Multi-Processing. In: **2008 IEEE International Symposium on Workload Characterization**. Washington, DC, USA: IEEE Computer Society, 2008. p. 35–46. (Cited on pages 12, 28, 54, 58, 68, 70, 71, 80, 81, and 82).

MOHAMMED, M. S.; ABANDAH, G. A. Communication characteristics of parallel shared-memory multicore applications. In: **2015 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)**. Washington, DC, USA: IEEE Computer Society, 2015. p. 1–6. (Cited on page 47).

MURURU, G.; GAVRILOVSKA, A.; PANDE, S. Quantifying and reducing execution variance in STM via model driven commit optimization. In: **2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)**. Washington, DC, USA: IEEE Computer Society, 2019. p. 109–121. (Cited on pages 28 and 29).

NETZER, R. H. B.; MILLER, B. P. What are race conditions?: Some issues and formalizations. **ACM Lett. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 1, n. 1, p. 74–88, mar. 1992. ISSN 1057-4514. (Cited on page 17).

NICÁCIO, D.; BALDASSIN, A.; ARAÚJO, G. LUTS: A lightweight user-level transaction scheduler. In: **Algorithms and Architectures for Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 144–157. (Cited on page 39).

NICÁCIO, D.; BALDASSIN, A.; ARAÚJO, G. Transaction scheduling using dynamic conflict avoidance. **International Journal of Parallel Programming**, Springer US, New York, NY, USA, v. 41, n. 1, p. 89–110, jul 2012. (Cited on pages 34 and 39).

OGASAWARA, T. NUMA-aware memory manager with dominant-thread-based copying GC. In: **Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications**. New York, NY, USA: ACM, 2009. (OOPSLA '09), p. 377–390. ISBN 978-1-60558-766-0. (Cited on page 45).

ÖZSU, M. T.; VALDURIEZ, P. Distributed and parallel database systems. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 28, n. 1, p. 125–128, mar. 1996. ISSN 0360-0300. (Cited on page 22).

PAPADIMITRIOU, C. H. The serializability of concurrent database updates. **J. ACM**, ACM, New York, NY, USA, v. 26, n. 4, p. 631–653, oct. 1979. ISSN 0004-5411. (Cited on page 23).

PASQUALIN, D. P.; DIENER, M.; DU BOIS, A. R.; PILLA, M. L. Online sharing-aware thread mapping in software transactional memory. In: **2020 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. Washington, DC, USA: IEEE CS, 2020. p. 35–42. (Cited on page 20).

PASQUALIN, D. P.; DIENER, M.; DU BOIS, A. R.; PILLA, M. L. Thread affinity in software transactional memory. In: **19th Int. Symposium on Parallel and Distrib. Comput. (ISPDC)**. Washington, DC, USA: IEEE CS, 2020. p. 180–187. (Cited on page 20).

PASQUALIN, D. P.; DIENER, M.; DU BOIS, A. R.; PILLA, M. L. Characterizing the sharing behavior of applications using software transactional memory. In: WOLF, F.; GAO, W. (Ed.). **Benchmarking, Measuring, and Optimizing (Bench 2020)**. Cham: Springer International Publishing, 2021. (Lecture Notes in Computer Science, v. 12614), p. 3–21. (Cited on page 20).

PELLEGRINI, F. Static mapping by dual recursive bipartitioning of process architecture graphs. In: **Proceedings of IEEE Scalable High Performance Computing Conference**. Washington, DC, USA: IEEE Computer Society, 1994. p. 486–493. (Cited on page 68).

PEREIRA, M. M.; AMARAL, J. N.; ARAÚJO, G. Measuring effective work to reward success in dynamic transaction scheduling. In: **2014 43rd International Conference on Parallel Processing**. Washington, DC, USA: IEEE Computer Society, 2014. (Cited on page 36).

PEREIRA, M. M.; BALDASSIN, A.; ARAÚJO, G.; BUZATO, L. E. Transaction scheduling using conflict avoidance and contention intensity. In: **20th Annual International Conference on High Performance Computing**. Washington, DC, USA: IEEE Computer Society, 2013. p. 236–245. ISSN 1094-7256. (Cited on page 40).

POPOVIC, M.; KORDIC, B. PSTM: Python software transactional memory. In: **2014 22nd Telecommunications Forum Telfor (TELFOR)**. Washington, DC, USA: IEEE Computer Society, 2014. p. 1106–1109. (Cited on page 41).

POPOVIC, M.; KORDIC, B.; BASICEVIC, I. Transaction scheduling for software transactional memory. In: **2017 IEEE 2nd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)**. Washington, DC, USA: IEEE Computer Society, 2017. (Cited on page 41).

POPOVIC, M.; KORDIC, B.; POPOVIC, M.; BASICEVIC, I. Online algorithms for scheduling transactions on Python software transactional memory. **Serbian Journal of Electrical Engineering**, National Library of Serbia, Beograd, Serbia, v. 16, n. 1, p. 85–104, 2019. ISSN 2217-7183. (Cited on page 41).

POUDEL, P.; SHARMA, G. Adaptive versioning in transactional memories. In: GHAFFARI, M.; NESTERENKO, M.; TIXEUIL, S.; TUCCI, S.; YAMAUCHI, Y. (Ed.). **Stabilization, Safety, and Security of Distributed Systems**. Cham: Springer International Publishing, 2019. p. 277–295. ISBN 978-3-030-34992-9. (Cited on pages 28 and 29).

RADOJKOVIĆ, P.; CAKAREVIĆ, V.; VERDÚ, J.; PAJUELO, A.; CAZORLA, F. J.; NEMIROVSKY, M.; VALERO, M. Thread assignment of multithreaded network applications in multicore/multithreaded processors. **IEEE Transactions on Parallel and Distributed Systems**, Los Alamitos, CA, USA, v. 24, n. 12, p. 2513–2525, Dec 2013. ISSN 1045-9219. (Cited on page 45).

RANE, A.; BROWNE, J. Performance optimization of data structures using memory access characterization. In: **2011 IEEE International Conference on Cluster Computing**. Washington, DC, USA: IEEE Computer Society, 2011. p. 570–574. (Cited on page 47).

RAVICHANDRAN, K.; PANDE, S. F2C2-STM: Flux-based feedback-driven concurrency control for STMs. In: **2014 IEEE 28th International Parallel and Distributed Processing Symposium**. Washington, DC, USA: IEEE Computer Society, 2014. p. 927–938. ISSN 1530-2075. (Cited on page 36).

RAYNAL, M. The mutual exclusion problem. In: **Concurrent Programming: Algorithms, Principles, and Foundations**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. chp. 1, p. 3–13. (Cited on page 18).

REINDERS, J. **Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism**. Sebastopol, CA, USA: O'Reilly Media, 2007. ISBN 0596514808. (Cited on page 43).

RIBEIRO, C. P.; CASTRO, M.; MÉHAUT, J.-F.; CARISSIMI, A. Improving memory affinity of geophysics applications on NUMA platforms using Minas. In: PALMA, J. M. L. M.; DAYDÉ, M.; MARQUES, O.; LOPES, J. a. C. (Ed.). **High Performance Computing for Computational Science – VECPAR 2010**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 279–292. ISBN 978-3-642-19328-6. (Cited on page 43).

RIBEIRO, C. P.; MEHAUT, J.; CARISSIMI, A.; CASTRO, M.; FERNANDES, L. G. Memory affinity for hierarchical shared memory multiprocessors. In: **2009 21st International Symposium on Computer Architecture and High Performance Computing**. Washington, DC, USA: IEEE Computer Society, 2009. p. 59–66. ISSN 1550-6533. (Cited on page 43).

RICO, T. M.; PILLA, M. L.; DU BOIS, A. R.; DUARTE, R. M. Energy consumption and scalability evaluation for software transactional memory on a real computing environment. In: **2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)**. Washington, DC, USA: IEEE CS, 2015. p. 7–12. (Cited on page 47).

RIEGEL, T.; FELBER, P.; FETZER, C. A lazy snapshot algorithm with eager validation. In: **Proceedings of the 20th International Conference on Distributed Computing**. Berlin, Heidelberg: Springer-Verlag, 2006. (DISC'06), p. 284–298. ISBN 3-540-44624-9, 978-3-540-44624-8. (Cited on pages 25, 27, and 28).

RIEL, R. V.; FENG, S. **Documentation for /proc/sys/kernel/**. 2020. <https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#numa-balancing>. Last accessed 18th, November 2020. (Cited on pages 110 and 111).

RITO, H.; CACHOPO, J. ProPS: A progressively pessimistic scheduler for software transactional memory. In: SILVA, F.; DUTRA, I.; COSTA, V. S. (Ed.). **Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings**. Cham, Switzerland: Springer International Publishing, 2014. p. 150–161. ISBN 978-3-319-09873-9. (Cited on page 36).

RITO, H.; CACHOPO, J. Adaptive transaction scheduling for mixed transactional workloads. **Parallel Comput.**, Elsevier Science Publishers B. V., Amsterdam, Netherlands, v. 41, n. C, p. 31–49, jan. 2015. ISSN 0167-8191. (Cited on page 40).

RUAN, W.; LIU, Y.; SPEAR, M. STAMP need not be considered harmful. 9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2014). co-located with ASPLOS 2014. <http://transact2014.cse.lehigh.edu/ruan.pdf>. 2014. (Cited on pages 70 and 77).

RUAN, W.; VYAS, T.; LIU, Y.; SPEAR, M. Transactionalizing legacy code: An experience report using GCC and memcached. In: **Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems**. New York, NY, USA: Association for Computing Machinery, 2014. (ASPLOS '14), p. 399–412. ISBN 9781450323055. (Cited on page 28).

RUGHETTI, D.; DI SANZO, P.; CICIANI, B.; QUAGLIA, F. Machine learning-based self-adjusting concurrency in software transactional memory systems. In: **2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems**. Washington, DC, USA: IEEE Computer Society, 2012. p. 278–285. ISSN 2375-0227. (Cited on page 40).

RUGHETTI, D.; DI SANZO, P.; CICIANI, B.; QUAGLIA, F. Analytical/ML mixed approach for concurrency regulation in software transactional memory. In: **2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing**. Washington, DC, USA: IEEE Computer Society, 2014. p. 81–91. (Cited on page 40).

RUGHETTI, D.; DI SANZO, P.; CICIANI, B.; QUAGLIA, F. Dynamic feature selection for machine-learning based concurrency regulation in stm. In: **2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing**. Washington, DC, USA: IEEE Computer Society, 2014. p. 68–75. ISSN 1066-6192. (Cited on page 40).

SAINZ, D.; ATTIYA, H. RELSTM: A proactive transactional memory scheduler. 8th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2013). 2013. Available from Internet: <http://transact2013.cse.lehigh.edu/sainz.pdf>. (Cited on page 38).

SASONGKO, M. A.; CHABBI, M.; AKHTAR, P.; UNAT, D. ComDetective: A lightweight communication detection tool for threads. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: ACM, 2019. (SC '19). ISBN 9781450362290. (Cited on page 30).

SCHERER III, W. N.; SCOTT, M. L. Advanced contention management for dynamic software transactional memory. In: **Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 2005. (PODC '05), p. 240–248. ISBN 1-58113-994-2. (Cited on pages 24 and 28).

SHARP, C.; BLEWITT, W.; MORGAN, G. Resolving semantic conflicts in word based software transactional memory. In: SILVA, F.; DUTRA, I.; COSTA, V. S. (Ed.). **Euro-Par 2014 Parallel Processing**. Cham, Switzerland: Springer International Publishing, 2014. p. 463–474. ISBN 978-3-319-09873-9. (Cited on page 38).

SHARP, C.; MORGAN, G. Hugh: A semantically aware universal construction for transactional memory systems. In: **Proceedings of the 19th International Conference on Parallel Processing**. Berlin, Heidelberg: Springer-Verlag, 2013. (Euro-Par'13), p. 470–481. ISBN 978-3-642-40046-9. (Cited on page 38).

SHAVIT, N.; TOUITOU, D. Software transactional memory. In: **Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 1995. (PODC '95), p. 204–213. ISBN 0-89791-710-3. (Cited on page 22).

SHAVIT, N.; TOUITOU, D. Software transactional memory. **Distributed Computing**, Springer-Verlag, Berlin, Heidelberg, v. 10, n. 2, p. 99–116, Feb 1997. ISSN 1432-0452. (Cited on page 25).

SOOMRO, P. N.; SASONGKO, M. A.; UNAT, D. BindMe: A thread binding library with advanced mapping algorithms. **Concurrency and Computation: Practice and Experience**, Wiley, v. 30, n. 21, p. e4692, jun. 2018. (Cited on pages 55 and 68).

SPEAR, M. F.; DALESSANDRO, L.; MARATHE, V. J.; SCOTT, M. L. A comprehensive strategy for contention management in software transactional memory. In: **Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. New York, NY, USA: ACM, 2009. (PPoPP '09), p. 141–150. ISBN 978-1-60558-397-6. (Cited on page 24).

SUTRA, P.; MARLIER, P.; SCHIAVONI, V.; TRAHAY, F. Boosting transactional memory with stricter serializability. In: SERUGENDO, G. D. M.; LORETI, M. (Ed.). **Coordination Models and Languages**. Cham, Switzerland: Springer International Publishing, 2018. p. 231–251. ISBN 978-3-319-92408-3. (Cited on page 25).

TAM, D.; AZIMI, R.; STUMM, M. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In: **Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007**. New York, NY, USA: ACM, 2007. (EuroSys '07), p. 47–58. ISBN 978-1-59593-636-3. (Cited on page 46).

TAY, Y. C. **Analytical Performance Modeling for Computer Systems: Third Edition**. 3rd. ed. San Rafael, California, USA: Morgan & Claypool Publishers, 2018. ISBN 1681733919, 9781681733913. (Cited on page 34).

THIBAULT, S.; NAMYST, R.; WACRENIER, P.-A. Building portable thread schedulers for hierarchical multiprocessors: The Bubblesched framework. In: **Proceedings of the 13th International Euro-Par Conference on Parallel Processing**. Berlin, Heidelberg:

Springer-Verlag, 2007. (Euro-Par'07), p. 42–51. ISBN 3-540-74465-7, 978-3-540-74465-8. (Cited on page 45).

TRAHAY, F.; SELVA, M.; MOREL, L.; MARQUET, K. NumaMMA - NUMA MeMory Analyzer. In: **Proceedings of the 47th International Conference on Parallel Processing - ICPP 2018**. New York, NY, USA: Association for Computing Machinery, 2018. (ICPP 2018). (Cited on page 44).

TRONO, J. A. Transactions: They're not just for banking any more. **J. Comput. Sci. Coll.**, Consortium for Computing Sciences in Colleges, USA, v. 30, n. 5, p. 160–166, may 2015. ISSN 1937-4771. (Cited on pages 17 and 18).

UNAT, D.; DUBEY, A.; HOEFLER, T.; SHALF, J.; ABRAHAM, M.; BIANCO, M.; CHAMBERLAIN, B. L.; CLEDAT, R.; EDWARDS, H. C.; FINKEL, H.; FUERLINGER, K.; HANNIG, F.; JEANNOT, E.; KAMIL, A.; KEASLER, J.; KELLY, P. H. J.; LEUNG, V.; LTAIEF, H.; MARUYAMA, N.; NEWBURN, C. J.; PERICAS, M. Trends in data locality abstractions for HPC systems. **IEEE Transactions on Parallel and Distributed Systems**, Institute of Electrical and Electronics Engineers (IEEE), v. 28, n. 10, p. 3007–3020, oct 2017. (Cited on page 19).

VILLAVIEJA, C.; KARAKOSTAS, V.; VILANOVA, L.; ETSION, Y.; RAMIREZ, A.; MENDELSON, A.; NAVARRO, N.; CRISTAL, A.; UNSAL, O. S. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In: **2011 International Conference on Parallel Architectures and Compilation Techniques**. Washington, DC, USA: IEEE Computer Society, 2011. ISBN 9780769545660. (Cited on page 46).

WANG, Z.; BOVIK, A. C. Mean squared error: Love it or leave it? a new look at signal fidelity measures. **IEEE Signal Processing Magazine**, IEEE Computer Society, Washington, DC, USA, v. 26, n. 1, p. 98–117, 2009. (Cited on page 59).

WONG, C. S.; TAN, I.; KUMARI, R. D.; WEY, F. Towards achieving fairness in the Linux scheduler. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 42, n. 5, p. 34–43, jul. 2008. ISSN 0163-5980. (Cited on pages 29 and 69).

YEH, T.-Y.; PATT, Y. N. Alternative implementations of two-level adaptive branch prediction. In: **Proceedings of the 19th Annual International Symposium on Computer Architecture**. New York, NY, USA: ACM, 1992. (ISCA '92), p. 124–134. ISBN 0-89791-509-7. (Cited on page 39).

YOO, R. M.; LEE, H.-H. S. Adaptive transaction scheduling for transactional memory systems. In: **Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures**. New York, NY, USA: ACM, 2008. (SPAA '08), p. 169–178. ISBN 978-1-59593-973-9. (Cited on pages 24, 34, and 35).

YU, Z.; ZUO, Y.; ZHAO, Y. Convoider: A concurrency bug avoider based on transparent software transactional memory. **International Journal of Parallel Programming**, Springer Science and Business Media LLC, v. 48, n. 1, p. 32–60, Sep 2019. ISSN 1573-7640. (Cited on pages 28 and 29).

ZHOU, N.; DELAVAL, G.; ROBU, B.; RUTTEN, É.; MÉHAUT, J. F. Autonomic parallelism and thread mapping control on software transactional memory. In: **2016 IEEE International Conference on Autonomic Computing (ICAC)**. Washington, DC, USA: IEEE Computer Society, 2016. p. 189–198. (Cited on page 24).

ZHOU, N.; DELAVAL, G.; ROBU, B.; RUTTEN, E.; MÉHAUT, J.-F. An autonomic-computing approach on mapping threads to multi-cores for software transactional memory. **Concurrency and Computation: Practice and Experience**, John Wiley & Sons, Ltd, Hoboken, New Jersey, EUA, v. 30, n. 18, p. e4506, may 2018. (Cited on pages 41 and 48).

Appendices

APPENDIX A SHARING-AWARE DATA MAPPING IN STM

This appendix shows how STMap (Section 6.2) can be extended to include sharing-aware **data mapping**. We opted for including the experiments on data mapping in the appendix since the main focus of this thesis is on sharing-aware **thread mapping**. The goal of data mapping is to optimize the usage of memory controllers, by mapping memory pages to the same NUMA node where the core that is accessing them belongs. To summarize, thread mapping aims to avoid access to the memory, prioritizing caches. On the other hand, if access to the memory is necessary, data mapping tries to map the memory that needs to be accessed to a local NUMA node, avoiding remote accesses.

A.1 Algorithm

To perform a sharing-aware data mapping in STM, it is necessary to know which NUMA nodes are accessing each memory page address. For this purpose, we follow the same intuition of the mechanism to detect thread communication, proposed in Section 6.2.3, with some modifications, shown in Figure 30 and detailed in Algorithm 5.

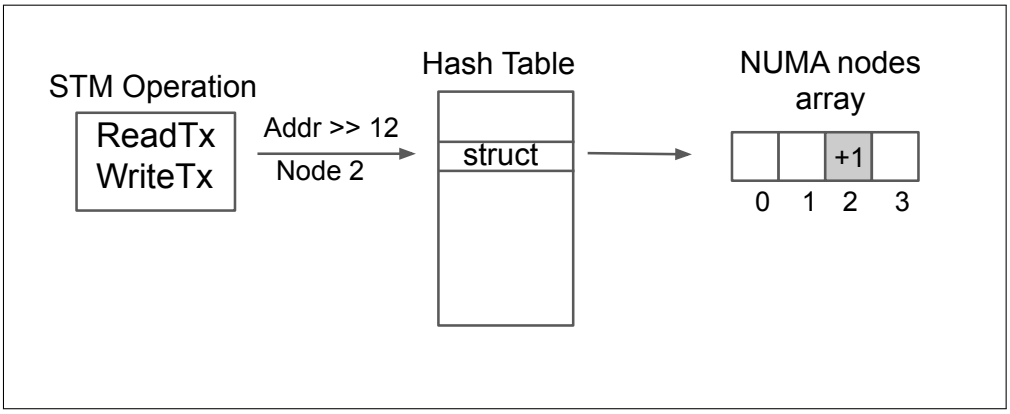


Figure 30 – Mechanism for detecting page accesses. Data structures are shown for a NUMA machine with 4 nodes (0-3).

Lines 1, 2 and 3 are exactly the same of the Algorithm 4 (Section 6.2.3), i.e., it is used to verify if it is time to sample the accessed memory page, based on the value of

Algorithm 5 Detecting memory pages accesses and performing data mapping.

Require:
addr: memory address being accessed
node: NUMA node that is accessing the memory page
tid: thread ID of the thread that is accessing the address
addr_sample : thread private variable used to determine if is time to sample the memory address
total_addr : thread private variable used to determine if is time to trigger the thread mapping
si: sample interval. Default 100
dmi: data mapping interval.
PAGE_SIZE_BITS: 12 for page size of 4096 bytes

```
1:  addr_sample ← addr_sample + 1
2:  if (addr_sample > si) then
3:      addr_sample ← 0
4:      pageaddr ← addr >> PAGE_SIZE_BITS                    ▷ Right shift
5:      elem ← getPageInfo(pageaddr)
6:      if (!elem.moved) then   ▷ Verify if the memory page already have been moved
7:          elem.nodes[node] ← elem.nodes[node] + 1   ▷ Increase the amount of access
8:  if (tid = 1) then
9:      total_addr ← total_addr + 1
10: if (tid = 1) and (total_addr ≥ dmi) then
11:     Compute new data mapping
12:     dmi ← dmi * 2
```

sampling interval. Since the STM runtime has access to the full memory address, we first need to bit shift the address to get the information of the memory page (line 4). To keep track of accessed memory pages, a hash table is used whose keys are memory pages. Each position of the hash table contains a structure with the memory address and an array of size equals to the NUMA nodes of the machine (Figure 30). Each position of this array contains the number of accesses to the memory page performed by each NUMA node. Hence, on line 5, the function `getPageInfo` gets from the hash table the structure containing information about the memory page being accessed. To avoid unnecessary page moves, we only update the number of accesses to this page (line 7) if the page not already have been moved (line 6). In summary, instead of calling Algorithm 2 (Chapter 4) to detect the sharing behavior between threads, the modified algorithm keep track of the number of times which each NUMA node is accessing a specific memory page (lines 4–7).

The lines 8 to 10 are the same of the Algorithm 4 (Section 6.2.3), keeping track of the amount of memory address accessed, in order to trigger the data mapping. On the line 11 the data mapping is triggered. This step is simpler than thread mapping: we verify on the hash table each memory page that not have been moved and which

NUMA node has most accessed it. Then, while the application is running, we send these information to the function `move_pages` of the `libnuma` library (KLEEN, 2004) to perform the page move.

Contrary to thread mapping, where our previous study showed that `STAMP` applications do not present dynamic sharing behavior (Chapter 5), hence, not being necessary to perform more than one time the thread mapping, on data mapping we cannot disable the mechanism after the first trigger. For thread mapping, just keeping track of a few memory accesses it is possible to deduce, with a certain level of accuracy, the memory access pattern of the application. For data mapping, it is not possible to deduce the future memory pages that will be accessed and which NUMA nodes will access them. However, to avoid a high overhead of moving pages, after each data mapping, on line 12, we double the next *data mapping interval* (dmi).

A.2 Improving STM applications with Data Mapping

Similar to the experiment made on Section 2.4 for thread mapping, we create an experiment with a synthetic array sum application. This application consists of an array of 2^{30} integer elements. In that case, the array uses approximately 4 Gigabytes of memory. We force the array to be initialized with zeros in the main thread. Hence, using the default *first touch* police, all memory will be allocated in the NUMA node of the main thread. To force the use of more than one NUMA node, the application was executed using 64 threads. The objective of this application is very simple. Each thread iterates through their array part thirty times. On each iteration, it updates the respective array position, incrementing the current value by one. We implemented the proposed mechanism of this Appendix inside the `TinySTM` and used it for synchronization of shared variables used in the array. Hence, the STM runtime will be aware of all the memory addresses that belong to the array. We iterate through the array thirty times to guarantee that if a memory page is migrated then it will be accessed again in the appropriate NUMA node, regarding the locality of the access.

The default Linux kernel already has routines to improve the memory page balancing of NUMA nodes. It keeps track of the page faults, moving the page automatically to the node that most accessed it. This mechanism is called *NUMA balancing* (RIEL; FENG, 2020). To test our proposal, we executed the array sum application in the Xeon and Opteron machines, described in Section 6.1.1. For comparison we used the following configurations:

- **Linux-NBOff** is the default Linux CFS scheduler, however with the NUMA balancing mechanism disabled.
- **Linux-NBOn** is the default Linux CFS scheduler with the NUMA balancing mech-

anism enabled. This approach will be useful to verify if the application is suitable for data mapping, or if the default *first-touch* approach is more efficient.

- **STMap** is the mechanism proposed in Section. 6.2. However, since we are interested in verifying the benefits of data mapping together with thread mapping, we do not use the heuristic proposed in Algorithm 3, hence, the thread mapping is always triggered one time.
- **STMap+DM** in this approach, we first trigger the thread mapping one time. After that, we begin to keeping track of the memory pages being accessed, triggering the first data mapping on *dmi* of 100,000 addresses such as in thread mapping (Sect 6.2).

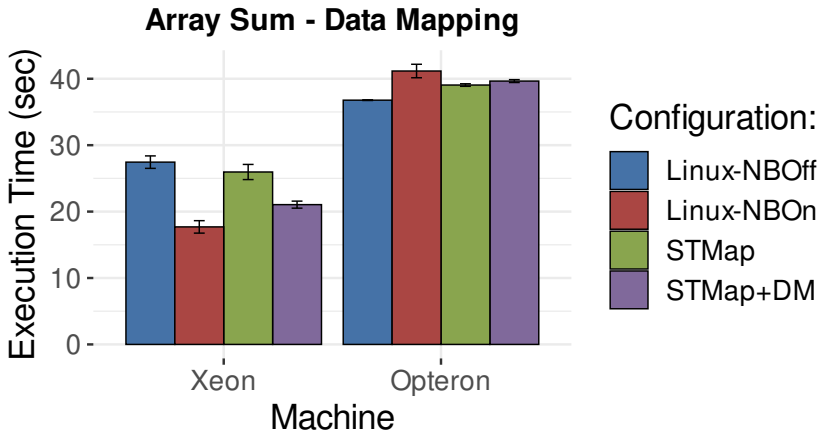


Figure 31 – Execution time of the Array Sum application.

It is worth noting that the *NUMA balancing* (RIEL; FENG, 2020) was enabled only in **Linux-NBOn** approach. In all other approaches, this mechanism was disabled. Figure 31 show the results. In the Xeon machine, it is possible to observe that NUMA balancing (**Linux-NBOn**) reduced the execution time by 35.5% when compared to the same mechanism without the balancing (**Linux-NBOff**). This proves that this synthetic application has an unbalanced memory page allocation. Although it was not possible to beat NUMA balancing, our proposed **STMap+DM** mechanism achieved performance gains of 23.3% when compared to **Linux-NBOff** and, 19% when compared to **STMap**. However, when analyzing the results in the Opteron machine, it is not possible to see any benefits of the data mapping. In fact, NUMA balancing decreases the performance by 12% when compared to **Linux-NBOff**.

As related in Section 6.1.1 the information about NUMA nodes distances gathered with `numactl` (KLEEN, 2004) indicates that the nodes distances in the Xeon machine are three times more distant than Opteron. This could explain in part the lack of results for data mapping in Opteron. To be sure about this information, we used a tool[1] to

[1] <https://github.com/matthiasdiener/numafac>

calculate the NUMA factor of the two machines, regarding bandwidth and latency of remote accesses. This tool uses `Lmbench` (MCVOY; STAELIN, 1996) to calculate the latencies. The NUMA factor is defined as the latency between memory accesses to remote NUMA nodes divided by the latency to access local nodes (MARIANO et al., 2016). Table 10 show the results. Although the bandwidth of the Xeon machine is

Table 10 – NUMA factor of the machines used in the experiments.

		NUMA factor	
Machine	**Node distances**	**Bandwidth**	**Latency**
Xeon	50 – 65	1.96	6.90
Opteron	16 – 22	1.28	2.16

larger than Opteron, the latency of remote accesses is three times slower in Xeon. This could explain in part why the array sum application only presented results on the Xeon machine. Nevertheless, in Section A.4 we will test the proposed data mapping mechanism using more realistic workloads.

A.3 Methodology

Although the experiments made on Section A.2 with the array sum application were not conclusive regarding the Opteron machine, we executed all eight benchmarks from `STAMP` and the two micro-benchmarks (HashMap and Redblacktree) using the proposed data mapping mechanism. The `STAMP` applications represent realistic workloads, being more appropriate to verify the efficiency of the proposed mechanism. We also added two more mechanisms to the comparison:

- **DM-100K** in this approach the thread mapping is not used, only data mapping, triggering the first mapping on 100,000 addresses.
- **DM-50K** this approach is used to verify if a more aggressive data mapping is better, triggering the first mapping on 50,000 addresses, i.e., half of the previous approach.

A.4 Results

The results of sharing-aware data mapping in both machines were similar. Hence, we will not discuss each application individually. Alternatively, we calculated the average speedup using **Linux-NBOff** as a baseline. Also, the results were grouped by thread number and machine. Figure 32 show the results.

Overall, the proposed mechanism does not improve the performance of STM applications. The best speedup was achieved by STMap, i.e., only triggering thread mapping,

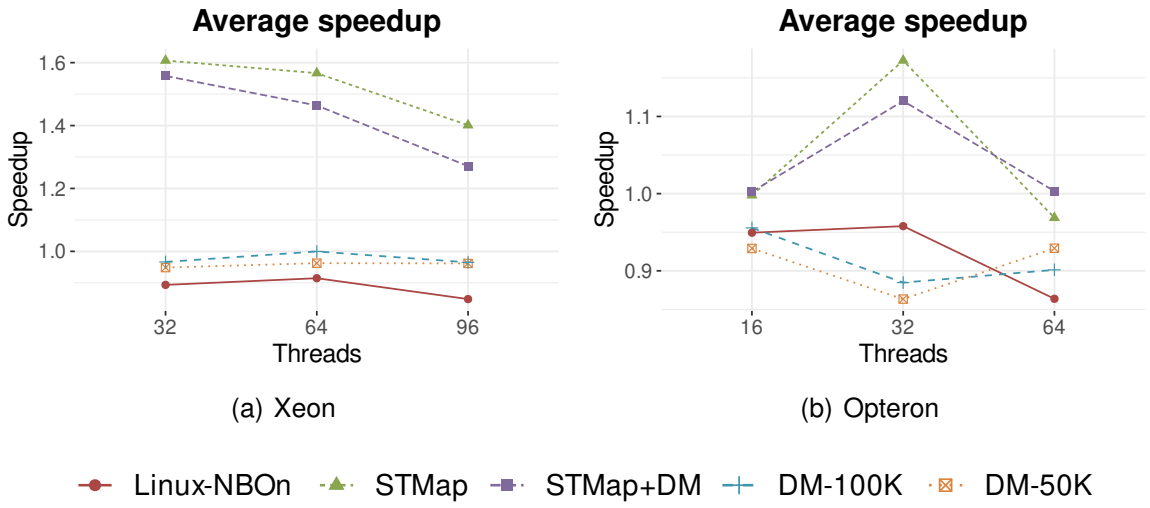


Figure 32 – Average speedup of the mappings when compared to Linux-NOff.

with exception of Opteron in 64 threads. When the data mapping was enabled together with thread mapping (**STMap+DM**) it decreased the performance. In that case, the overhead of the data mapping mechanism was not compensated by the better exploration of the locality of memory pages. However, analyzing the speedup, overall, on both machines the NUMA balancing mechanism (**Linux-NBOn**) decreased the performance. On the Xeon machine, all proposed mechanisms performed better than NUMA Balancing. In that case, the default *first-touch* police were more efficient. Hence, initially, two hypotheses can be formulated. The first is that the applications utilized in the experiments are not suitable for data mapping. The second is that the proposed mechanism failed in exploring data mapping. However, this second hypothesis was tested in Sect A.2 and showed results in the Xeon Machine.

A.5 Discussion

Albeit the proposed mechanism in this appendix does not improve the performance of the STM applications used in the previous chapters, a synthetic application showed that in specific scenarios it can improve the performance. More specifically, it is necessary an STM application with atomic blocks that protect a large amount of distinct variables and a NUMA machine with high latency in remote node accesses.

Since STM runtime has precise information about shared variables, this information can be used to choose which threads should be mapped closer to each other to share caches, i.e., it is not necessary a global vision of the application sharing behavior (Chapter 6). However, for an efficient data mapping, it is necessary a global vision of the memory pages of the application, not only the ones accessed by the STM runtime. In the synthetic array sum application presented in Sect A.2, the STM runtime was aware of all the 4 Gigabytes of the array. Nevertheless, data mapping showed improvements

only in the Xeon machine, with high latencies on remote node access.

Using more realistic workloads, we do not see performance improvements. Probably, in the majority of STM applications, the STM runtime is not aware of the entire memory pages of the application.

A.6 Summary

This appendix proposed an extension to STMap to include sharing-aware data mapping. The extension keeps track of the number of accesses of each NUMA node to each memory page. On each data mapping interval, the memory page is moved to the NUMA node that most accessed it.

Using a synthetic array sum application, it was showed that the proposed mechanism could increase the performance on NUMA machines with high latency on remote memory access. However, using more realistic workloads, it was not possible to improve the performance. Our thoughts are that it could be infeasible to perform a sharing-aware data mapping take into consideration only STM operations because it represents only a fraction of the memory pages used by the entire application.