Multi-dimensional Self-tuning in Transactional Memory

Ricardo Moreira Freire da Silva Neves

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson: To Be Defined
Supervisor: Prof. Paolo Romano
Member of the Committee: Prof. José Costa

October 2015
Acknowledgments

I want to thank my advisor Professor Paolo Romano, for all the support and attention provided throughout the Thesis.

Also, I want to thank Diego Didona, and Nuno Diegues, for their continuous support and collaboration through all steps of my research.

Lastly, I would like to thank Shady Aala, who have supported me at the beginning of my research.

This work was partially supported by FCT GreenTM (EXPL/EEI ESS/0361/2013).
Abstract

Transactional Memory is a powerful abstraction that promises to drastically simplify parallel programming. However, the efficiency of existing TM implementations can be strongly affected by the characteristics of the workloads generated by TM applications. This has raised interest in designing self-tuning solutions that adapt specific building blocks of existing TM implementations. This dissertation first overviews the state of the art in the area of TM, analyzing the key trade-offs that affect the performance of these systems. Next it discusses existing self-tuning solutions for TM, highlighting a critical limitation of existing approaches: these only support the dynamic adaptations of individual building blocks of a TM, which prevents existing solutions from pursuing globally optimal configurations. In the light of this critical analysis, in this thesis I present ProteusTM, the first TM with multi-dimensional self-tuning. The system encompasses in its core a large library of state of the art TMs, along with dedicated modules to: monitor performance, adapt its internal implementation, and predict optimal configurations. Each of these configurations consists in a tuple belonging to a \( N \) dimensional space where each dimension corresponds to a TM parameter, e.g., TM backend, parallelism degree. To achieve the desired optimal level of performance, the predictive module relies in a novel technique based on Collaborative Filtering and Bayesian Optimization. The extensive evaluation conducted in ProteusTM showed that it delivers an average performance that is only 3% away from optimal, and gains up to 100% over static alternatives.

Keywords: Parallel Computing , Self-tuning , Machine Learning , Transactional Memory
Resumo

Memória Transacional é uma poderosa abstração que promete simplificar drasticamente a computação paralela. Contudo, a eficiência das existentes implementações de MT são fortemente afetadas pelas características das cargas de trabalho de aplicações de MT. Tal, despertou o interesse no desenho de soluções auto-ajustáveis que adaptam blocos específicos de implementações de MT. Esta dissertação primeiro providencia uma visão geral sobre o estado da arte na área de MT, analisando as implicações chave que afetam o desempenho destes sistemas. De seguida, discute-se as soluções auto-ajustáveis existentes na área de MT, destacando a limitação crítica das abordagens atuais: estas apenas suportam a adaptação dinâmica de blocos individuais de MT. Tal, previne as soluções de procurarem configurações globalmente ótimas. Face a esta análise crítica do estado da arte, nesta tese apresento ProteusTM, a primeira MT com capacidades multi-dimensionais. O sistema engloba no seu núcleo uma larga biblioteca das TMs do estado da arte, juntamente com módulos dedicados para: monitorizar o desempenho, adaptar-se internamente, e para prever configurações ótimas. Cada uma destas configurações consiste num tuplo que pertence a um espaço com N dimensões, em que cada dimensão corresponde a um parâmetro de MT, e.g., Biblioteca de MT, grau de paralelismo. Para alcançar o deseado nível de desempenho ótimo, o módulo preditivo baseia-se numa nova técnica baseada em Collaborative Filtering e Bayesian Optimization. A extensa avaliação realizada em ProteusTM, mostrou que este tem um desempenho em média apenas 3% afastado do óptimo, e ganhos até 100% comparado com alternativas estáticas.

Keywords: Computação Paralela, Auto-ajustamento, Aprendizagem automática, Memória Transacional
Contents

Acknowledgments iii

Abstract v

Resumo vii

1 Introduction  2

1.1 Contributions .................................................. 3

1.2 Structure of the document ...................................... 5

2 Background 7

2.1 Transactional Memory ........................................... 7

2.1.1 Software Transactional Memory ............................ 8

2.1.2 Hardware Transactional Memory ......................... 14

2.1.3 Hybrid Transaction Memory ............................... 17

2.2 Self-tuning ..................................................... 19

2.2.1 Supervised Learning ....................................... 21

2.2.2 Tuning the TM parameters ................................. 22

2.2.3 Thread-Mapping ............................................. 24

2.2.4 Contention Manager ....................................... 25

2.2.5 Tuning the TM backend .................................... 26

2.2.6 Distributed TM self-tuning ................................. 28

2.3 Collaborative Filtering in Recommender Systems .......... 28
List of Tables

3.1 Machines used in the TM parameters study. .......................... 34
3.2 DFO avg., median, and 90th percentile for the CM variants. ........ 38
3.3 Percentage of sampling with DFO lower than 25%, 50%, and 75%, for each number of threads. .................................................. 42
3.4 Percentage of sampling with DFO lower than 25%, 50%, and 75%, for each TM backend. ......................................................... 43
4.5 TM applications used in the evaluation. These 15 benchmarks span a wide variety of workloads and characteristics. ..................... 54
4.6 Machines used in the experimental test-bed. ........................... 55
5.7 Parameters tuned by ProteusTM. STMs are TinySTM [46], SwissTM [43], NORec [26] and TL2 [34]. .................................................. 60
5.8 Overhead (%) incurred by ProteusTM for different TM and # threads. Results are an average across ten runs. ................................. 60
5.9 Reconfiguration (TM and #threads) latency (µsec). ..................... 61
5.10 For each benchmark (of Fig. 5.15), it is showed the MDFO (in %) of ProteusTM, each Optimal and BFA (★) configurations. Each workload is labeled with its optimal configuration. ........................................... 68
List of Figures

3.1 Performance heterogeneity in TM applications. ........................................ 35
3.2 Win percentage of each CM variant. ....................................................... 37
3.3 CDF of DFO for throughput. ................................................................. 38
3.4 System with CM vs System without CM - CDF of DFO for throughput. .......... 39
3.5 Win percentage for each number of threads. ......................................... 40
3.6 CDF of DFO for throughput. ................................................................. 41
3.7 Win percentage of each TM backend. .................................................... 43
3.8 CDF of DFO for throughput. ................................................................. 44
4.9 Architecture of the ProteusTM system. .................................................. 47
4.10 Switching TM algorithm safely in PolyTM. ............................................ 48
5.11 Rating distillation for Exec. Time on Machine A. .................................. 62
5.12 Controller’s exploration policies for EDP on Machine A (left figures) and Exec. Time on Machine B (right figures). .................................................. 63
5.13 Comparing early-stop exploration predicates. ........................................ 64
5.14 Comparison vs Machine Learning based techniques. .............................. 65
5.15 Performance of four applications when their workload changes three times. It is shown the performance obtained with ProteusTM, and three additional fixed configurations, each one corresponding to an optimum in each workload. ............ 66
5.16 Similar to Fig. 5.15c, but with a static application workload, varying instead the availability of machine resources. ............................................. 68
Chapter 1

Introduction

Over the last years, processors manufacturers faced a problem with increasing the clock speed of a single processor. As a result, parallel processor architectures have entered the realm of mainstream computing: nowadays commodity processors adopt a multi-core organization, according to which computing capacity is expanded by increasing the number of cores, rather than enhancing the processor’s clock speed.

However, having more cores is not equivalent to having a single more powerful CPU. In fact, in order to take advantage of the computing capacity offered by multi-core architectures, application developers are required to embrace a major paradigm shift, moving from sequential to parallel programming. Unfortunately, parallel programming is notoriously more challenging. One well-known, critical problem of parallel programming is how to synchronize concurrent accesses to shared data. In fact, the conventional approach to synchronization is based on locking, which is well known to suffer of severe issues like lack of composability, deadlocks, and livelocks [62].

Given the huge relevance that parallel programming has gained over the last years, the academic and industrial research community have invested a great effort to identify a synchronization mechanism that could represent a simpler, yet efficient alternative to locking. Currently, Transactional Memory (TM) is a prominent proposal in this sense [44, 41].

TM brings the familiar abstraction of transactions, used for decades with success in the area of database systems, to the domain of parallel programming. By requiring programmers to only specify what should be executed atomically, and not how atomicity should be achieved, TM can drastically simplify the problem of correctly synchronizing concurrent computations. Further, by leveraging on scalable and efficient concurrency control mechanisms, possibly accelerated via dedicated hardware supports, TM has been shown to deliver performance comparable, or even
superior, to the one achievable by complex and error-prone fine-grained locking mechanisms [41, 94].

On the down side of the coin, the design space of a TM implementation is huge, and existing research has shown that no-one-size-fits-all design exists that can deliver optimal performance for all possible workloads [19, 39, 41, 91, 94]. To further complicate the matter, algorithmic designs are not the only factor influencing TM performance. The correct tuning of the parameters of a TM algorithm, as well the characteristics of the hardware platform are known to have a strong impact on the efficiency of existing TM libraries.

These considerations motivate the research on self-tuning mechanisms capable of dynamically adapting the various internal mechanisms encompassed by TM systems, in order to match the characteristics of application’s workloads as well as of the underlying hardware infrastructure.

This thesis addresses this critical issue by proposing ProteusTM, a TM with self-tuning capabilities, with the purpose of building a high performance adaptive TM system while maintaining the same ease of programming.

1.1 Contributions

The main contribution of this thesis is ProteusTM, the first TM with multi-dimensional self-tuning capabilities. ProteusTM contains two main modules:

- **PolyTM** is a polymorphic TM library that encapsulates state-of-the-art results from research in TM, and has the unique ability to transparently and dynamically adapt across multiple dimensions: (i) switch between different TM algorithms; (ii) reconfigure the internal parameters of a TM; (iii) adapt the number of threads concurrently generating transactions.

- **RecTM** is in charge of determining the optimal TM configuration for an application. Its basic idea is to cast the problem of identifying such best configuration as a recommendation problem [74]. This allows RecTM to inherit two highly desirable properties of state of the art Recommender System (RS) algorithms: the ability to operate with very sparse training data, and to require only the monitoring of the Key Performance Indicator (KPI) to be optimized. This avoids intrusive instrumentation [79] and (possibly inaccurate) static code analysis [92] employed by other machine learning-based solutions.

While building ProteusTM, several challenges were solved:

- **Transparency and portability**: PolyTM encapsulates a wide variety of TM implementations, along with their corresponding tuning procedures. The key challenge here is to conceal these
mechanisms without breaking the clean and simple abstraction of TM. Furthermore, one of the key design goals of ProteusTM is to seamlessly integrate with existing TM applications, and to support different machine architectures.

This issue was tackled by integrating PolyTM in GCC, via the standard TM ABI [70], and by exposing to programmers standard C++ TM constructs. Not only this preserves the simplicity of the TM interface, but it also maximizes portability due to the widespread availability of GCC across architectures.

- **Minimizing the cost of adaptivity.** Supporting reconfiguration across multiple dimensions requires introducing some degree of synchronization, in order to ensure correctness during runtime adaptations. The challenge here is to ensure that the overheads to support adaptivity are kept small enough not to compromise the gains achievable via self-tuning.

This challenge was addressed by designing lightweight synchronization schemes that exploit compiler-aided, and asymmetric code instrumentation. The combination of these techniques allows PolyTM to achieve a negligible overhead of around 1% and a maximum overhead of 8%, even when considering the most performance sensitive TM implementations.

- **Applying Recommender Systems to the TM domain:** Decades of research have established RS as a powerful tool to perform prediction in various domains (e.g., music and news) [64, 31, 28]. The application of RS techniques to performance prediction of TM applications, however, raises unique challenges, which were not addressed by previous RS-based approaches to the optimization of systems' performance [33, 32]. One key issue here is that, in conventional RS domains (e.g., recommendations of movies), users express their preferences on a homogeneous scale (e.g., 0 to 5 stars). On the contrary, the absolute value of key performance indicators (KPIs) of TM applications can span very heterogeneous scales. As it will be discussed, this can severely hinder the accuracy of existing RS techniques.

To cope with this issue, a novel normalization technique has been introduced, called rating distillation, which maps heterogeneous KPI values to scale-homogeneous ratings. This allows ProteusTM to leverage state-of-the-art RS algorithms even in the presence of TM applications whose KPIs' scales span across different orders of magnitude.

- **Large search space:** Although RS algorithms are designed to work with very sparse information, their accuracy can be strongly affected by the choice of the configurations [86] that are initially sampled to characterize a TM application. Deciding which and how many TM configurations to sample is a challenging task, as ProteusTM supports reconfiguration across multiple dimensions, resulting in a vast search space.
1.2. STRUCTURE OF THE DOCUMENT

RecTM addresses this issue by relying on Bayesian optimization techniques [9] in order to steer the selection of the configurations included in the characterization of a TM application. This reduces by up to $4 \times$ the duration of the learning process of the RS using Collaborative Filtering (CF) [86].

1.2 Structure of the document

The remaining of the document is organized as follows.

Chapter 2 provides background on and an overview of work related to two areas: Transactional Memory and Machine Learning techniques applied to the self-tuning of adaptive systems. As so, this chapter covers all the relevant research conducted in TM: its origins, the broad design space, and current state of the art implementations. Also, it contains a section dedicated to self-tuning, which illustrates the motivations behind it and the most prominent techniques to implement it. In particular, the section focuses on in Machine Learning techniques, and Collaborative filtering from the Recommender Systems field of study, which plays a major role in the developed solution.

Chapter 3 contains a study aimed at inferring the sensitivity of TM performance to the tuning of several internal parameters. It focuses on identifying which TM parameters have the most crucial impact on performance. In particular, we consider a parametric space of three dimensions (TM implementation, degree of parallelism, contention management) and study how the individual and joint variation of these parameters impact both performance and energy efficiency. One interesting result of this preliminary study is that the tuning of one of the four considered dimensions (contention management) provides marginal benefits, if one can tune optimally the remaining two parameters. This result is exploited in the remainder of this dissertation, as it allows for reducing the dimensionality of the self-tuning problem that we tackle in the following, while designing ProteusTM.

Chapter 4 represents the core of the thesis, as it presents the architecture of ProteusTM. In particular, it describes i) how ProteusTM efficiently supports the reconfiguration of internal parameters in a transparent fashion to the running application ii) how ProteusTM detects shifts in the workload generated by the application, so as to trigger the re-optimization of the TM and iii) how ProteusTM identifies the optimal configuration for the TM in a multi-dimensional and large search space.

Chapter 5 reports and analyzes the results of an extensive evaluation of ProteusTM. In particular, this chapter focuses on two aspects: i) it demonstrates the minimal overhead incurred by ProteusTM to support adaptivity, and ii) it shows that ProteusTM outperforms state-of-the-art
solutions to the same and related problems.

Finally, chapter 6 concludes this dissertation, by resuming the key results achieved and discussing possible directions for future work.
Chapter 2

Background

This section provides background information on two main areas: Transactional memory and Self-tuning.

Section 2.1 is devoted to provide background on TM. More in detail, it overviews its origin and motivations. Next, it introduces Software, Hardware and Hybrid TM implementations, discussing the motivations and differences for each kind of TM, as well as the respective state of the art implementations for each.

Section 2.2 focuses on self-tuning. First, some fundamental methodologies used in this domain are introduced, with an emphasis on the techniques that are going to be employed in the remainder of this dissertation. Next, a survey on existing self-tuning TM systems is provided.

Lastly, section 2.3 focuses on Collaborative-Filtering. It first describes the basis of this technique, and after it presents the background behind Matrix Factorization and Bayesian Optimization.

2.1 Transactional Memory

Transactional Memory is a lightweight concurrency control mechanism that many researchers believe to be the path to follow to achieve performance comparable (and under some conditions better) to fine-grained locking, providing at the same time the ease of use of coarse-grained locking.

As mentioned before, with TM the programmer only needs to specify a critical zone of code as atomic, and the underlying implementation will take care of correctly synchronizing it, removing loads of complexity from the programmer’s shoulders.
This mechanism is based on the concept of transactions, which was successfully introduced decades ago in the context of database systems. As in classic database transactions environments, TM transactions preserves atomicity; as such, a transaction either has no effect (i.e., it is aborted), or appears to take effect instantaneously in some point in time (serialization point) within its start and completion.

Transaction memory enables several transactions to run in parallel with or without locks. The detection of invalid operations that would break consistency can be performed either while a transaction is running (eager) or at commit-time (lazy). This ensures that a given transaction has a consistent view of the shared memory.

TM was initially proposed as an extension for multi-processors cache-coherency protocol [55]. However, the lack of architectural support for transactions led researchers to focus on software-based solutions to further investigate the potentiality of the approach. Moreover, hybrid solutions have been proposed, which try to reconcile the software and the hardware.

2.1.1 Software Transactional Memory

Software Transactional Memory is the most common implementation of TM, since its implementation is not bound to the availability of any architectural support [10, 27, 47, 35, 48, 49].

At their basis, STM implementations rely on special data structures that track what a transaction read and write (read and write set). These sets are checked (either eagerly or lazily) to detect conflict.

Throughout the years, many STMs have been proposed, with the purpose of improving its performance. From the base design choices exploited, it is possible to identify the following alternatives:

1. word-based vs object-based - granularity level at which memory is accessed. Word-based means that the memory is accessed at the granularity of machine words or larger chunks of memory. Object-based implies that accesses to memory are done at object granularity and it requires the TM to be aware of the object associated in every access.

2. lock-based vs lock-free - whether or not a TM resorts to locks to correctly handle concurrency.

3. write-back vs write-through - how updates are written to memory. Write-through writes the updates directly to memory and previous values are stored in an undo log, providing lower commit-time overhead than Write-back. On the other hand, Write-back writes updates to memory upon commit, and stores it in a write log, granting lower abort overhead than the
latter.

4. encounter-time locking vs commit-time locking - when conflict detection is performed either at commit (lazy) or during execution (eager).

As it was demonstrated, there exists a several number of different design choices, and internal parameters configurations to apply, and these decisions have a huge impact on the TM performance.

The best design and configuration can vary according to the architecture, depending on the CPU and the size of the caches. More important to note is that the efficiency that will result from these choices will hugely vary depending on the workload generated by the application.

Most of the researchers agree that there is no STM implementation that can always deliver the best performance independently of the workload [21, 39, 47, 91].

In this subsection, three STM implementations will be analyzed, namely TinySTM, JVSTM, and NOrec. These STMs explore different possibilities of the design space for what concerns the aforementioned design choices. Such different design choices, as hinted in Section 1, result into these STMs being optimized for different kinds of workload: TinySTM provides high scalability on write-intensive workloads, NOrec was conceived to minimize the overhead at low thread counts, and JVSTM excels at long-running read-only transactions.

It will be given focus to the design choices made by the authors of each one, as well as the auxiliary mechanisms incorporated, which led to these different specializations.

2.1.1.1 NOrec

No ownership records is an STM library presented in 2010, by Luke Dalessandro, Michael F. Spear, and Michel L. Scott [27].

NOrec is mostly known for being a TM that incorporates some desirable features (e.g., low fast-path latency, publication and privatization safety, livelock freedom, etc.).

NOrec is specifically designed to incur very low overhead at low thread counts, at the cost of exhibiting limited scalability. Its design is based on a low overhead STM algorithm, presented by Spear et al. named TML [85].

TML uses a single global sequence lock, which has the advantage (over traditional reader-writer lock) that readers are invisible, and induce the coherence overhead of updating the lock. It uses eager conflict detection and in-place updates, where writes acquire the lock for writing, and reads check the lock version to ensure consistency.
TML scales poorly, since it uses a single global lock with eager conflict detection, meaning that only one writer can be active at a time.

NOrec design diverges from TML, in the sense that it tends to improve scalability over the latter. While TML is eager, NOrec uses a write-back strategy to write updates to memory, and performs conflict detection at commit-time.

The lazy conflict detection for concurrent speculative writers, enables to minimize the quantity of time that a writer holds the lock. Such, increases the probability that concurrent read-only transactions will commit. Reads are validated eagerly, in order to avoid a transaction to read inconsistent values and prevent erroneous behavior in transactions that are destined to abort.

Privatization problem [84] arises when an object is accessed by transactional and nontransactional code at the same time (e.g., private to a nontransactional thread), i.e., TM systems must avoid violation of atomicity, consistency and isolation in such scenarios to assure safety.

The serialization of writeback ensures privatization safety [84], and combined with value-based validation, ensures publication safety[66]. This allows for accessing shared data structures outside transactions, with no additional cost, as long as the program is transactional data race free.

It does also support closed nesting [69], enabling to abort an inner transaction without making the outer parent to immediately abort as well.

This STM also has high compatibility to legacy systems, and the authors think that would be a viable choice to use as a fallback software support for hardware, or hybrid TMs.

2.1.1.2 SwissTM

SwissTM is a TM software library proposed by Dragojévic et al. in 2009 [44].

It attempts to achieve higher effectiveness than the other state of the art TMs, by relying on both eager and lazy acquisition techniques.

Similar to TL2, and TinySTM it is a word based TM, which resorts to redo-logging in order to support both acquisition schemes.

At its basis, this STM relies on a timestamp that every transaction possesses, which is assigned when a transaction begins, grabbing the value from a global counter. Upon a successful read, a transaction always updates its timestamp with the current value of the global counter.

When writing to memory, a transaction first confirms if she is the owner of the writing lock for that address. If that is the case, the transaction updates the value of the address in its write log and returns. If that is not the case, the transaction first tries to acquire the lock, through a
2.1. TRANSACTIONAL MEMORY

compare-and-swap (CAS). If CAS returns false, the transaction either rollbacks, retries, or waits for the owner of the lock to finish.

Upon validation, a transaction compares the current version of all the locations read with the version stored in the read-log. The validation fails if a mismatch is detected, otherwise, it succeeds.

In SwissTM, a read-only transaction can commit immediately (the consistency of the read log is assured). In the case of a transaction that contains writes, she first has to grab all locks of the locations written. After it, the transaction increments the commit timestamp and revalidates. In case of failure, it rollbacks, otherwise, the transaction updates the values of all written addresses as well as the commit timestamp, and lastly it releases the read and write locks.

The evaluation conducted by the authors suggests that SwissTM can outperform other state-of-the-art STM implementations such TL2, and TinySTM in some specific workloads.

2.1.1.3 TinySTM

TinySTM is a STM implementation presented by Pascal Felber, Christof Fetzer and Torvald Riegel in 2008 [47].

This TM is a word-based implementation that resorts to locks to protect shared memory locations, and to an encounter-time locking (eager) scheme to guarantee that any transaction always read consistent states. It also allows the co-existence of both write-through and write-back designs for memory access.

The authors justify the adoption of encounter-time locking with the two following considerations:

1. Their experimental observations seem to indicate that if the conflicts are detected early, then the transaction throughput will increase because there will not be useless work done by transactions.

2. It allows to handle read-after-writes efficiently without requiring expensive mechanisms. Encounter-time locking enables a transaction to temporarily acquire the locks corresponding to the memory locations that it accesses. In combination with write-through it assures that the memory always contains the latest value written.

The validation of transactions at commit-time is implemented using a Hierarchical Locking scheme. This strategy uses a shared array of \( l \) locks, and a smaller “hierarchical” array of \( h \ll l \) counters. Choosing \( l \) as a multiple of \( h \),

\[
    l = 2^i, h = 2^j, i > j
\]  

(2.1)
it is possible to compute the index of the lock and the counter, for a given address:

\[
lockIndex(addr) = hash(addr) \mod (l)
\]

\[
\text{counterIndex}(addr) = hash(addr) \mod (h)
\]

This scheme allows transactions to determine whether locks have been acquired, and can be generalized to multiple levels of nesting.

TinySTM's internals are optimized for workload that exhibit the following characteristics:

1. update transactions read many locations.
2. few writes from concurrent transactions.

The authors tested, and believe that these conditions are often found in real applications, to consider it a useful strategy.

\subsection*{2.1.1.4 TL2}

Transactional Locking 2 is an STM proposed by Dice et al. in 2006 [35].

In its basis it relies in a combination of commit-time locking and a global version-clock based validation technique.

The idea behind the implemented algorithm is the use of a global version-clock, which is incremented for each transaction that performs writes to memory, and is read by all transactions.

In this STM, all memory locations have a lock associated that contain a version. Transactions start by reading the global clock, and then every location read is validated against it, guaranteeing that only consistent memory views are read at a low cost.

When writing, transactions need to collect a read-set. Upon validation, the transactions acquire the locks of the locations written, increment the global clock, and validate the read-set as an attempt to commit. After committing, the transactions update the memory locations with the new global version-clock, and the respective locks are released.

TL2 was presented before the STMs from the previous subsections. Still, nowadays, it keeps being recursively used as baseline to compare against.

When TL2 was presented it solved important STM challenges, among those, it guaranteed that user code operates only on consistent memory states. TL2 managed to provide this new property while delivering performance as good or better than the former state of the art STMs.
2.1. TRANSACTIONAL MEMORY

2.1.1.5 JVSTM

João Cachopo, and António Rito Silva in 2006 [10].

It is actually used in a production environment on the IST’s FénixEDU project as a replacement of the previous lock-based concurrency control mechanism. JVSTM ensures that object cache maintained by the middle-tier servers are consistently and atomically updated with the back-end database [15, 77].

The original version of this STM used commit-time locking with a single global lock, nested transactions, and versioned boxes.

A nested transaction is a transaction (child) that starts within the context of another transaction (parent). A versioned box is a container that keeps a sequence of values, and can be seen as a replacement for memory locations, and transactional variables.

It was the first known STM in literature to guarantee that read-only transactions would never conflict with concurrent ones, which improved the concurrency on applications consisting of long transactions containing considerably more reads than writes.

To minimize the overhead of running applications with a large fraction of read-only transactions (common scenario in many workloads) [11], JVSTM's default configuration speculates that any starting transaction is going to be read-only. This avoids tracking the transaction’s read set and improve performance in case the speculation is correct. If the transaction is detected to not be read-only, it is re-executed.

JVSTM employs two techniques to further reduce conflicts:

1. Delaying computations - avoids high-contention boxes, and re-executes the parts of the transaction that caused the conflicts. To effectively use this technique, it uses boxes that hold values private to a transaction.

2. Restartable transactions - allows the parts that caused the conflict to be re-executed.

Like stated before, the original version of JVSTM [10] used a global lock to handle mutual exclusion at commit-time. Later in 2011, Sérgio Miguel Fernandes and João Cachopo published a second, enhanced, implementation of JVSTM, which adopts a lock-free scheme [48].

The change was motivated for some reasons, e.g.: Poor performance in applications containing transactions that execute several writes in a short period. Thus, high contention on the global lock will significantly degrade performance.

The new commit algorithm proposed uses some concepts of the old one, e.g., read-set validation,
write-back, commit visible to other transactions. While the original lock-based used snapshot validation as a validation technique, this new version uses incremental validation. The latter, consists in checking every write-set committed since the transaction started, in search for an intersection with its read-set.

The size of the write-set is application dependent, hence the list to iterate at validation depends on the number of write transactions, which hinders the incremental validation performance. To minimize this issue, Fernandes et al. [48] introduced a small modification to the validation procedure combining snapshot with incremental.

Before any validation, a transaction helps to write-back all pending commits already in a queue, returning the last record that it helped to commit, which is assigned to a variable named lastSeenCommited. After that, a snapshot validation is performed (without synchronization, being possible that concurrent commits occur). However, the set of committed versions for each transactional location contains at least all the commits up to the version of the lastSeenCommited record, performing validation up to that point.

One last validation still needs to be performed, checking for any newer commits, to grant a valid commit. This will be done through incremental validation, but only from the lastSeenCommited. This algorithm is slower for low transaction counts, but scales better because it keeps the list of write-sets small.

### 2.1.2 Hardware Transactional Memory

Hardware Transactional Memory is a more recent bet to build a superior TM.

In HTM, transactions are executed with the aid of specific architectural support and processor instructions.

Although the first TM proposal was hardware, the absence of HTM support in commercial processors, limited most of researchers to emulators to test their ideas, leading to a major focus on STM to advance the state of art.

Recently, both Intel and IBM implemented support for HTM on some of their new processors. For this reason, real-world HTM implementations are garnering much attention in the last years.

Like the Herlihy and Moss’s original TM proposal [55], these TMs rely on modified cache coherence protocols in order to achieve atomicity and isolation.

The main advantage of HTM over STM is that it requires no instrumentation comparing to STM solutions. Since it does not require to instrument reads and writes, it avoids most of the overheads
incurred by STMs that could severely degrade the performance. On the other hand, a major drawback of HTM lies in the fact that it cannot guarantee that a transaction will ever succeed (even without concurrency) due to its limited nature, which has to keep the read-write set into L1 cache.

Taking this into account, one of the major concerns is to develop an efficient software fallback mechanism as an alternative synchronization mechanism to use whenever HTM restrictions may prevent transactions from ever committing.

Intel acknowledged that programmers must provide this software fallback path on the begin instruction, deciding in this way what should be done upon abort.

Example of restrictions are: long transactions, which contains lots of operations and can exceed the size of the L1 cache; interrupts; and context switches.

On the rest of this section, the two topics below will be addressed:

1. TSX
2. BlueGene/Q

It will be presented a brief overview of both HTM implementations (Intel TSX, IBM BlueGene/Q), and results of some evaluation done so far.

### 2.1.2.1 IntelTSX

Intel started including supports for HTM with the release of their Haswell (4rd gen.) processors, in 2012, and nowadays it is deployed on several machines, from tablets to servers.

The hardware support was made available by extending the instruction set for x86 with Transactional Synchronization Extensions (TSX), which is the first generation of Intel's commodity HTM. TSX provides two interfaces: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM).

HLE allows optimistic execution of a critical section through eliding the acquisition of a lock when writing, so it appears as free to other threads.

TSX supports HLE by the use of two operates (XAQUIRE and XRELEASE) that can be placed in LOCK instructions. In Haswell processors those allow to elide the lock, assuring that it will be read but not written, enabling concurrent threads to execute the same section simultaneously. Data races are prevented by keeping track of the speculative accesses, rolling back the execution if the accesses made are invalidated by cache coherency. In such case, the thread re-execute
the section but this time the lock is acquired and released normally. Concurrent elisions of the same lock are aborted, since the lock is part of their transactional footprint.

RTM offers more flexibility, offering instructions such as \texttt{XBEGIN} and \texttt{XEND}, that map directly to the usual construction beginning and committing of a transaction. Another advantage is that it allows the application of more fallback strategies upon abort of the hardware transaction, unlike HLE that gives up immediately. On the other hand, determining the best fallback strategy is not an easy task, as it ultimately depends on the workload.

The provided hardware support is best-effort nature, and, due to inherent HTM limitations, it cannot guarantee if a transaction will ever be able to commit, not ensuring progress guarantees. Intensive evaluations done on Intel TSX were recently published by Diegues et al. [41]. The evaluation was done comparing this HTM to several state of arts STMs, and coarse/fine-grained locking schemes. Among other conclusions, the study demonstrates that TSX can clearly outperform any synchronization scheme in some workloads (e.g., short and infrequent transactions) but performs poorly in others (e.g., long and contention-prone transactions, where STM and fine-grained locking perform better).

\subsection{2.1.2.2 IBM Blue Gene/Q}

Blue Gene/Q is one of the first implementations of HTM for commercial processors, and it was made available by IBM.

Being a HTM it partially solves some of the main problems of STM solutions, such as: the high overheads from starting and committing a transaction, and the instrument and monitoring of memory references inside a transaction. But like expected from its nature, it does not ensure forward progress, and like Intel TSX it can benefit from software support to improve its performance.

The BG/Q provides the following to support transactional execution:

1. \textbf{Buffering of speculative states} - writes made during a transaction execution store a speculative state, that is buffered on the L2 cache, and becomes visible to the other threads after commit, in an atomic way.

2. \textbf{Hardware conflict detection} - hardware detects write-read, read-write, and write-write between concurrent transactions, or when a transaction is followed by a non-transactional access to the same address.

Hardware support is primarily implemented on the L2 cache that acts as a point of coherence, and efficiently supports TM enabling a considerably large speculative state.
2.1. TRANSACTIONAL MEMORY

The HTM alone shows good results on several benchmarks [90]. Unfortunately, there are no extensive works that compare Blue Gene with STMs. However, due to its HTM best effort nature, it is likely to exhibit limitations and trade-offs similar to the TSX case. Like the Intel TSX an efficient software support is desirable to assure forward progress.

2.1.3 Hybrid Transactional Memory

Hybrid Transactional Memory is a TM that enables both hardware and software transactions to be executed in conjunction within the same TM.

After the first real HTMs like TSX and BlueGene/Q were released, an augmented interested on HyTMs solutions emerged. Due to the best-effort nature of HTMs, it is desirable to find a scalable STM to be used in the fallback path to ensure progress, instead of coarse lock-based schemes. HyTM schemes aim to deliver the best of both TM types: HTMs efficiency and modern STMs scalability. Some TM researchers argue that HyTM has plenty of potential to be exploit [12, 25, 41, 63], but there is still lots of work to be done.

One of the latest results worth to mention on this field is Invyswell, a HyTM released this year, which will be addressed in detail on this section.

2.1.3.1 Invyswell

Invyswell is a recent published HyTM presented by Calciu et al., in 2014 [12]. This HyTM is a scalable hybrid implementation that uses a HTM along with a STM fallback, to guarantee forward progress.

It uses hardware transactions from Intel RTM, and software transactions from a modified version of an STM called InvalSTM [49]. Like any HyTM, Invyswell pursues the objective of enabling both hardware and software transactions to be concurrently executed, with the objective of delivering a solution that is effective for all transactions sizes and contention levels.

The STM used provides scalability and performance for large transactions with considerable contention, and has a distinctive difference compared to other STMs, i.e., it performs commit-time invalidation [49].

Commit-time invalidation allows the STM to have full knowledge of all conflicts between a committing transaction and others in-flight transactions, allowing it to make decisions on how to best mitigate contention. This design choice is extremely useful on HyTMs as it allows to mitigate the conflicts between software and hardware transactions. Another important aspect, is that it allows
read-only transactions to commit without incurring serialization overhead, being transparent to RTM's faster executing hardware transactions.

This STM aims at optimizing performance against large transactions workload, complementing RTM, which excels with short transactions. Regarding this interplay, it is important to note that the STM chosen, does not serialize read-only transactions, enabling RTM to execute without interfering with the STM when the latter executes read-only transactions.

In this implementation, Haswell's hardware transactions are instrumented to track their read and write set via Bloom Filters. InvalSTM can then leverage on the information maintained by these bloom filters for detecting conflicts with concurrently executing hardware transactions.

To manage the shared-memory between HTM and STM, Invyswell performs the conflict detection between a hardware and a software transaction after the hardware transaction has committed. This is unavoidable, since any write operation performed by an hardware transaction is buffered until commit time (TSX does not support non-transactional operations from within a transactional context). RTM does not support escape actions, hence when a hardware transaction conflicts with a software one, it aborts. By combining invalidation and conflict detection after a hardware transaction, this scheme minimizes the chance to abort a hardware transaction due to an in-flight software one.

The need for assuring progress guarantees, and adaptability to different heterogeneous workloads, lead Invyswell to support five transactions types:

1. SpecSW - software speculative transaction that uses private Bloom filters to track accessed memory locations. Performs invalidation after committing, which makes it compatible with hardware transactions that can invalidate in-flight SpecSWs.

2. BFHW - hardware bloom filter-based transaction. Used to handle conflict with software transactions, by preventing software transactions (SpecSWs) from committing or reading, while is still committing. After commit, it performs post-commit invalidation, marking as invalid all in-flight SpecSWs.

3. LiteHW - lightweight hardware transaction executed without read or write annotations. Can only commit if there are no in-flight software at the beginning of their commit-phase. Optimal for small transactions.

4. IrrevocSW - direct update software transaction, which acquires the commit lock as soon as it execution begins. Used to assure progress guarantees.

5. SglSW - direct update software transaction that does not allow concurrent execution of other
2.2. SELF-TUNING

software transactions. Used for small transaction that contain instructions not supported by RTM, assuring progress guarantees.

The transactions are scheduled in a performance descending order: first the high-risk hardware transactions, then the low-risk software transactions. The transitions between the types are decided at runtime, based on an application independent heuristic.

Finally, Invyswell relies on a Contention Manager that can be used by software transactions, which decides how to handle conflicts based on information such as the priority of each conflicting transactions and the size of their read and write sets.

The experimentation done with Invyswell shows that this HyTM delivers a better performance on a range of STAMP benchmarks [67] than the state of the art NOrec[27], and the authors believe that the hybrid mechanisms can become a first choice when Haswell platforms with more cores are released.

2.2 Self-tuning

The success of a transactional memory largely depends on the fact that it greatly simplifies the task of writing and maintaining parallel applications. However, as previously shown, the performance of a TM strongly depends on the affinity between its design and the workload characteristics.

Taking into account such considerations, self-tuning is needed to deliver optimal performance across all workloads, by reconfiguring the TM at runtime depending on the currently workload characteristics. This mechanism, in an ideal scenario, can permit the TM system to predict changes, adapting the system in a dynamic way to the configuration that will provide the best performance.

A self-tuning scheme needs to answer the following:

1. When to adapt?
2. How to adapt?

Regarding when to trigger the adaptation, the self-tuning mechanism can either be designed to react to workload changes (reactive), or to try anticipate them (proactive).

Reactive schemes determine the need for reconfiguration based on the recent observations of the workload. Proactive schemes try to anticipate the need for reconfiguration by predicting future
workloads, and their effectiveness highly depends on the precision of the mechanisms chosen to predict these workloads.

Upon detection or prediction of the workloads changes, the self-tuning mechanism has to decide how to adapt to this change, i.e., which adaptation should be triggered, if any. The proper identification of the optimal configuration for a given workload is performed by the use of performance models, which allows the prediction of the system’s performance in the available configurations to trigger.

Couceiro et al. [19] recently classified the performance models that can be used in self-tuning systems, in three categories:

1. White-box modelling
2. Black-box modelling
3. Grey-box modelling

White-box modelling leverages on the available knowledge of the systems and applications, and codifies it into a model (e.g., analytical or simulative), to capture how the system configuration and workload characteristics translate to performance. These models typically do not require training, still can benefit from a minimal sampling phase to get value for some parameters. The drawback of white box is that it relies on approximations, and assumptions, that may hinder the performance in certain situations [19].

Black-box modelling does not require any prior knowledge on the internal dynamics of the target system, and it relies on a training phase, observing the inputs (e.g., workload characteristics) and outputs (e.g., KPIs like throughput or energy consumption) of a system, with the objective to infer a statistical model (e.g., based on ML techniques) that captures the observed relations between inputs and outputs. It relies in inferring the performance function by observing the output (e.g., throughput, read/writes ratio) corresponding to an input (e.g., workload with small read-only transactions, using a configuration for HTM to retry 5 times). In comparison to White-box, this modelling provides adaptability (e.g., initially unknown workloads), and tends to achieve a better precision, but could require a big amount of samples to build an efficient performance function.

Black-box can be further divided into off-line and on-line approaches. Offline approaches rely on a controlled training phase, during which the system experiments with different input/outputs to infer the performance function. Online approaches, instead, aim at finding a configuration that maximizes performance by exploring different configurations at runtime.

Grey-box modelling [38] is a hybrid approach that uses both white and black modelling, attempting to benefit from the best that the two can offer: minimal training (White-box), and enhanced
accuracy via retraining (Black-box). Existing grey-box techniques include ensembles of white and box learners [37] as well as approaches that use white-box models to initialize the knowledge of black-box learners [23].

With the available knowledge, the self-tuning implementation has to decide what should be the best decision for a system, for it to get the best possible performance. The literature on the self-tuning TM has explored a wide number of alternative techniques that are heterogeneous both in the employed modelling technique and in the target self-tuned aspect (e.g., TM parameters, TM switching, Thread Mapping Strategy, Contention Manager).

The remaining of this section begins with an introduction in Supervised learning, containing a brief description of the Black-box techniques used as baseline in the system’s evaluation (chapter 5). After this, it presents an overview of the mentioned self-tuning schemes: TM internal parameters tuning, Adaptive thread mapping, Contention Manager tuning, and TM switching. Lastly, it contains some honorable mention self-tuning techniques for Distributed TM.

### 2.2.1 Supervised Learning

Supervised Learning is a black-box modelling approach that has been highly employed in self-tuning. This learning methodology assumes the existence of a supervisor who can decide whether or not an agent is correct.

This type of learning relies on labeled data where for each row it contains the correct answer, which provides feedback throughout the learning process. As so, in this type of learning each row/example is a pair $<i, o>$, where $i$ is the input value (usually a vector) and $o$ is the answer labeled as correct, also called supervisory signal.

Supervised learning algorithms analyze the training set in order to infer a function/model that is capable of predicting the expected label for unknown data. Note that, if the codomain of $f$ is continuous, the learner is considered regressor. Otherwise, if the codomain is discrete, the learner is then a classifier, and its output values are termed classes.

Various approaches have been proposed in supervised learning. In the remainder, a brief description of several supervised learning algorithms is provided, focusing in particular on the ones that will be employed in the evaluation of ProteusTM (Chapter 5).

**Decision Trees (DT)**

DTS are tools that use a tree graph structure to support decisions. The nodes in the graph
represent tests, and its branches correspond to the outcome of each test. Lastly, the leaves represent the decision after traversing all nodes, which in the classifier case corresponds to a class.

**Artificial Neural Networks (ANN)**

ANN are a family of supervised learning algorithms that infer a function/model through a procedure analogous to biological neural networks. These tools are known to have been successfully applied to complex data with a high amount of inputs, even in scenarios in which a subset of the features are unknown. ANN can be seen as a system of interconnected "neurons" that exchange messages between each other, and are organized in two different layers, i.e., input and output layers. Each connection between neurons has a weight associated. With the experience gathered throughout the learning, the weights of the connections are adjusted.

**Support Vector Machines (SVM)**

SVM are machine learning methods that map the training data into a higher dimensional input space and constructs a precise separating hyperspace in this. The model represents the samples as points, mapped in such a way that samples of unrelated categories are divided by the hyperspace, and separated as wide as possible. The new data is mapped into the space, and predicted to which category it belongs based on the side of the hyperspace they fell. The mapping from the training data to the higher dimensional space is done by a function, called *Kernel*. Initially SVMs were introduced for classification purposes [18], but it has been extended to support the inference of a regressor model [5].

### 2.2.2 Tuning the TM parameters

Research on TM has shown that a variety of parameters have a huge impact in the delivered performance. The tuning of the internal parameters of a TM has been subject of study by various works [39, 47, 80, 81]. Again, the dynamic change of these parameters is motivated by the acknowledge of the TM research community that there is no configuration that can fit better than all the others independently of the workload characteristics.

An example of a mechanism that focus on tuning the retry policy of an HTM is Tuner. Tuner is a self-tuning mechanism for Intel TSX, presented by Nuno Diegues and Paolo Romano, in 2014 [39].

It is an innovative approach that makes use of learning techniques, and profiling at run-time in
2.2. SELF-TUNING

order to identify the optimal TSX configuration in a workload.

Diegues and Romano show that properly handling HTM retry parameter, can provide a huge boost on performance.

Tuner deals with optimizing the retry strategy of TSX by tuning two internal HTM parameters:

1. Number of retries before giving up to software

2. How to spend the number of retries

Regard point one, it has been recently reported that 5 appears to be the best all round value for the maximum number of attempts in hardware [39, 41, 94]. Still static configuration delivers sub-optimal performance, due to the heterogeneity of the workloads generated by applications.

As for point two, the budget of retries can decrease linearly, divide by two, or set immediately to zero upon detecting a capacity abort in a transaction.

An important feature, is that it enables to individual tune the parameters of each application’s atomic block, instead of using a single global configuration. This feature is relevant in programs that contain transactions with different characteristics, leading to heterogeneous workloads, benefiting from different configurations.

The final solution applied Upper Confidence Bounds (UCB) as a learning technique, considering each atomic block as a UCB instance. This technique estimates which one, among the available options for a given parameter, has the highest reward, and it is used to optimize the consumption of the attempts upon capacity aborts.

To optimize the configuration of the number of attempts for each atomic block, Gradient Descent Exploration (GRAD) is used, which is an exploration technique similar to hill climbing.

These two techniques work in conjunction in the final algorithm, with a hierarchy between the two, so they don’t overlap, allowing UCB to force GRAD to just explore in another direction, or avoiding "ping-pong" optimizations between the two [39]. The authors acknowledged that the joint strategy provided results that were always better than using only one of the approaches alone.

Still, in the context of self-tuning TM parameters, TinySTM has a scheme to tune the lock granularity.

This STM, which was already discussed at the STM Section, has a self-tuning mechanism aimed at optimizing the following parameters:
1. The hash function to map a memory location to a lock.

2. Number of locks.

3. Size of the array for the hierarchical locking.

The employed strategy keeps the throughput corresponding to each tested configuration, where a configuration is a triple formed by number of shifts, the number of locks, and the size of the hierarchical locking array.

It relies on a multi-dimensional hill climbing algorithm with memory, forbidden areas, and eight possible moves, as so, it works by making a move and then verifying its effectiveness during the next period.

### 2.2.3 Thread-Mapping

Multicore processors usually possess complex memory hierarchies of different levels of cache to reduce accesses to main memory. However, this can increase the time to access memory and degrade bandwidth usage if threads are not correctly placed on cores.

Thread-Mapping problem consist into allocating a thread onto the core that minimizes memory latency, and reduces memory contention through data locality.

Devising an optimal thread mapping strategy for a TM application is cumbersome because of two main causes:

1. TM applications are characterized by irregular behaviors, given by phased workloads and by its speculative nature.

2. Performance of a thread mapping strategy for a TM application not only depends on the target application, but also on the underlying TM backend.

One of the latest works on this matter focused on TM, was made by Castro et al.[16]. This study proposes four adaptive thread mapping strategies: two of which do not require any prior knowledge from TM applications.

In order to increase performance, the adaptive strategies must select the appropriate mapping for different TM applications, and STM configurations. The decision of the thread mapping is dynamic, since the workload may change during the execution. The adaptive thread mapping strategies require different information from the workload, to decide the mapping to apply, and are divided in two categories:
2.2. SELF-TUNING

1. Single metric-based

2. ML-based

Like the name suggests, the single metric based approach relies on monitoring at runtime a single metric to characterize the behavior of a workload and to perform the self-tuning choice. For this, the authors propose two adaptive thread mapping strategies making use of two reference metric: conflict level (Conflict), and the execution time (Test).

The ML-based approach, instead, leverages on an offline trained ML algorithm to drive the self-tuning process. The authors applied two ML algorithms to develop the following two ML-based strategies: ID3, i.e., a Decision Tree, and Apriori algorithm, i.e., an association rule learner.

Concluding this topic, this recent paper [16] shows that the performance of TM applications can be improved by better exploiting the memory hierarchy of multicores, through the use of Thread-Mapping, presenting different adaptive strategies to find the best thread mapping for TM applications. An important note, is that during the evaluation process, the authors found that on average ML-based adaptive strategies deliver a better performance than single metric-based ones.

2.2.4 Contention Manager

Most of STM systems guarantee weak progress property, named obstruction-freedom, which, consists in assuming that a transaction that runs for a long period without overlapping with any transaction will eventually commit [11, 50, 84].

Since this property does not exclude livelock or starvation, stronger properties can be provided by a module called contention manager (CM).

A CM is a mechanism that resolves conflicts between transactions, by deciding which transaction to abort and when and how to reschedule the execution of the aborted transaction. The CM affect liveness, not safety, and can be evaluated by the number of transactions committed per time.

Throughout the years, many different contention managers have been proposed [27, 12, 54, 83]. Evidence suggest that no CM outperforms all the others for every workload.

On the lights of this consideration, Guerraoui et al. proposed a self-tuning mechanism for TM, named Polymorphic Contention Manager [50], which, allows transactions to have different CMs, in a tentative to provide the CM that provides the best throughput for a given transaction according to the situation. After an extensive experimentation, the authors of this study, concluded that there is no CM that performs best independent of the circumstances.

Polymorphic CM is a module that allows not only to switch CMs across workloads, but also across
concurrent transactions in a single workload, and even between phases of a single transaction. One of the problems of mixing CMs is how can CMs of different classes interact in a useful way. A hierarchy of contention manager classes was implemented, taking into consideration the cost associated to each CM class, and generalizing groups of CM classes. The cost associated has to do, with the quantity of information that a CM keeps, being the less costly, those who do not keep any information, and the most costly those that maintain a lot of information about the current transaction, other transactions, etc.

A last note on the topic, is that this study goes one step further, by allowing nested transactions, and allowing different CMs for these.

2.2.5 Tuning the TM backend

As already discussed, the design of each TM implementation is optimized for a target workload. As such, there is no single TM implementation that delivers the best performance across all possible workloads. For this reason, there is a recent proposal by Wang et al. [91] that implements methods to construct policies with the available knowledge and dynamically choose the most appropriate TM that maximizes performance.

To select the best STM for a workload, this work relies on information gathered through static analysis and dynamic measurement. The adaptivity policy framework can be activated during program execution by four events:

1. number of consecutive aborts exceeds threshold,
2. long delays when attempting to begin a transaction,
3. thread creation and destruction,
4. commit rate below defined threshold.

The profiling triggered by the events above, uses a simple custom STM named by the authors as ProfileTM [91] to sample per-transaction characteristics.

When one of these conditions is met, a profiling phase is performed. The profiling process is the following: the library blocks new transactions, and waits for all in-flight transactions to either commit or abort. Then the TM is switched to ProfileTM, and N transactions are ran, one at a time. After profiling, the system changes the TM to the one recommended.

If the recommended is the same as the current active TM library, the system stores the total number of commits and aborts, until the next trigger of the framework. By storing this information,
2.2. SELF-TUNING

if the same TM is chosen again, then it will only remain if there has been forward progress, and the abort limit for causing another trigger will be doubled.

The adaptivity policies for selecting the most suitable algorithm, can either be created by a programmer, generated by an ML system, or by a collaborative process between the programmer and the learning tool.

The authors divided these policies in two general types: Expert policies and ML-based policies.

Expert policies are written by a programmer to satisfy some requirements, like for example the intuition that the best algorithm depends on the maximum number of active threads.

ML-based policies are automatically created through machine learning techniques.

This is the first paper in literature to present an ML-based adaptivity system for synchronizing TM programs. The experimentation results showed that ML-based adaptivity offers great performance, maintainability, and flexibility, making TM switching using policies automatically generated by ML, an attractive approach to maximize performance of TM applications, despite heterogeneity and complexity of those.

Although, this work only supports switching between STMs. The solution developed takes a leap further by aiming to support HTM too, since current HTMs outperform state of art STMs in some workloads [41, 94].

Still in the topic of TM Switching, to the issue of how to perform the switch, Lev et al. presented PhTM [63].

The main idea of this mechanism relates to being not safe to change modes without waiting for some conditions to be verified, i.e., in some cases it is better to delay changes. The delay allows to avoid trashing phenomena in which the TM is continuously switching between modes, resulting into poor performance.

The changing of TM backend (mode) is based on a single global variable named $modeIndicator$ that contains 6 fields, being the most relevant:

1. current mode
2. number of transaction that must complete before switching
3. next mode
4. number of transactions that will be switched to the next mode

In certain modes, some transactions cannot be completed, e.g., functionality is not supported.
To tackle this problem, a thread joining a mode can access the `modeIndicator`, changing the needed field, to assure correctness.

### 2.2.6 Distributed TM self-tuning

To conclude the topic of Self-tuning, I will briefly present two more works, morphR [21] and polycert [20].

Despite being focused on Distributed TM, they present interesting concepts, which will be applied on the system to develop (Section 3).

MorphR is a framework that supports generic adaptations, and introduces the concepts of fast-switching vs stop-and-go [21]. This can be extremely useful in a system composed by multiple reconfigurable components, where the reconfiguration process is a major part of the overall performance of the solution.

Fast-switching transitions are non-blocking and can be used when there is knowledge that both states can safely coexist, allowing for a faster reconfiguration process.

On the other hand, Polycert supports the coexistence of multiple protocol schemes of the same family [20].

Similar to what I will develop, this system relies on Oracles, which use forecasting in this case to determine the optimal certification scheme. The protocols used coexist naturally, since all of them rely on a common phase, during which they establish global serialization. Each message is tagged with a label that specifies which is being used for each transaction.

It is possible to create an analogy between tagged messages, and a possibility for tagged transactions, allowing for different TMs to coexist, something that could be explored.

Still, the more interesting aspect is the knowledge that grants that they can coexist, which applied with Morphr fast-switching, can enhance the process of reconfiguration.

### 2.3 Collaborative Filtering in Recommender Systems

A Recommender System (RS) seeks to predict the rating that a user would give to an item. These ratings can be exploited to recommend items of interest to users [64].

We focus in the following on CF [86], which is one of the most prominent prediction techniques used in the RS domain.
To infer the rating of a \langle user, item \rangle pair, CF techniques exploit the preferences expressed by other users, and ratings previously provided by the user on different items. Ratings are stored in a Utility Matrix (UM): rows represent users and columns represent items. Typically, a UM is very sparse, as a user rates a small subset of the items. A CF algorithm aims to reconstruct the full UM, from its sparse representation, by filling empty cells with ratings close to the ones that the users would give.

K Nearest Neighbors is perhaps the most intuitive CF approach: it tries to infer the rating for a given \langle user, item \rangle couple \langle u, i \rangle by finding users that have given ratings similarly to u, and/or items that have received ratings similar to i. The concept of similarity between two users/items is captured by a similarity function, which takes as input two rating arrays \( V_1, V_2 \) and outputs a real value in \([-1, 1]\).

The most widely employed similarity functions are the Euclidean, the Cosine and the Pearson. Note that two arrays may contain ratings for different set of elements, thus, similarity between the two arrays may rely on some computation done only on elements that the two vectors share; given two arrays \( V_1, V_2 \) of \( N \) elements each, \( C = \{ i \in V_1 \cap V_2 \} \) is defined as the set of indices of such common elements. The Euclidean function expresses similarity between two vectors as inversely proportional to their Euclidean distance:

\[
sim_E(V_1, V_2) = \frac{1}{1 + \sum_{i \in C} \sqrt{(V_1(i) - V_2(i))^2}}
\]

The Cosine distance expresses similarity between two vectors by computing the scalar product of the two and normalizing by the product of their norm; this represents the cosine of the angle between the two vectors:

\[
sim_C(V_1, V_2) = \frac{\sum_{i \in C} V_1(i)V_2(i)}{\sqrt{\sum_{j} V_1^2(j)} \sum_{k} V_2^2(k)} \tag{2.4}
\]

finally, Pearson similarity expresses similarity between two vectors as the correlation between the two, i.e., the extent to which they linearly relate with each other. Noting \( \overline{V} \) the average value of an element in \( V \):

\[
sim_P(V_1, V_2) = \frac{\sum_{i \in C}(V_1 - \overline{V_1})(V_2 - \overline{V_2})}{\sum_{k} \sqrt{(V_1 - \overline{V_1})^2} \sqrt{(V_2 - \overline{V_2})^2}}
\]

KNN-based CF approaches can search for similarities among users (user-based KNN) or/and among items (item-based KNN). In the user-based case, in its basic form, the prediction for a \langle u, i \rangle couple, noted \( R(u, i) \), is carried out in the following way: (i) find the set \( U_k \), composed by the \( K \) users that are most similar to \( u \) and that have rated \( v \); (ii) compute the prediction as the weighted average of the ratings expressed by users in \( U_k \) for \( v \), where the weights are computed
by the means of the similarity function:

\[ R(u, i) = \frac{\sum_{u' \in I_u} \text{sim}(u, u') R(u', i)}{\sum_{u' \in I_u} \text{sim}(u, u')} \]

In the item-based case, the prediction is computed in the following steps: (i) isolate the set \( I_u \) of items for which user \( u \) has already expressed a rating; (ii) identify the set of \( K \) items \( I_k \) whose ratings vector is the most similar to the vector associated with the item for which the prediction is being carried out; (iii) compute \( R(u, i) \) as the weighted average of the ratings expressed by \( u \) on items of \( I_k \):

\[ R(u, i) = \frac{\sum_{j \in I_k} \text{sim}(i, j) R(u, j)}{\sum_{j \in I_k} \text{sim}(i, j)} \]

### 2.3.1 Matrix Factorization

Rather than searching for similarities explicitly encoded in the user/item rating vectors, Matrix Factorization-based techniques try to identify latent patterns that are hidden in the Utility Matrix. Matrix factorization models map users and items to a joint latent factor space of dimensionality \( d \), such that user-item interactions are modeled as inner products in that space. Each dimension in this \( d \)-dimensional space represents a hidden similarity concept; taking as example the movies rating domain, a similarity concept may be how much a user likes drama movies and, correspondingly, how much a movie falls into the drama category.

Each user \( u \) is thus mapped onto a vector \( p_u \in \mathbb{R}^d \) and each item onto a vector \( q_i \in \mathbb{R}^d \); the prediction for an unknown \( \langle \text{user}, \text{item} \rangle \) couple is given by the inner product of the two corresponding vectors \( R(u, i) = q_i^T p_u \). Singular Value Decomposition (SVD) is an effective approach to identifying and extract latent semantic factors for CF [74]: it can be applied to factorize the Utility Matrix \( A \) in two smaller matrices \( P, Q \), such that \( Q^T P = A \).

However, SVD is conceived for factorizing matrices with no blank entries, whereas in the CF context the Utility Matrix is typically very sparse. A prominent approach to overcome this issue is to resort to Stochastic Gradient Descent (SGD) [6] in order to reconstruct a full Utility Matrix \( R \) that resembles as much as possible the sparse Utility Matrix \( A \).

Starting from two random \( Q \) and \( P \), in its basic form, SGD finds \( P \) and \( Q \) such that \( Q^T P = R \sim A \) by iteratively adjusting the values of elements in \( P \) and \( Q \) so as to minimize the square fitting error of \( R \) with respect to \( A \). Once \( Q \) and \( P \) are available, it is possible to estimate ratings via the inner product between the \( i \)–th item vector and the \( u \)–th user one: \( R(u, i) = q_i^T p_u \).
2.3. COLLABORATIVE FILTERING IN RECOMMENDER SYSTEMS

2.3.2 Bayesian Optimization

Bayesian Optimization (BO) [9] is a strategy for optimizing a target unknown function $f : D \rightarrow \mathbb{R}$. It has gained much attention for its capability to successfully optimize functions for which it is cumbersome or impossible to derive a closed expression and whose estimation can, thus, only be obtained through (possibly noisy) observation at sampled values. BO techniques have been shown to be among the most efficient approaches in terms of number of observations required to identify extreme of a function [9], thus resulting particularly suitable to optimize black-box functions whose direct evaluation is expensive. Sequential Model-based BO (SMBO) [57] is a recent development in the field of BO that operates according to the following steps: 

1. evaluate the target function $f : D \rightarrow \mathbb{R}$ at $n$ initial points $x_1 \ldots x_n$ and create a training $S$ set with the resulting $< x_i, f(x_i) >$ pairs; 
2. fit a probabilistic model $M$ over $S$; 
3. use a so called acquisition function $a(M,S) \rightarrow D$ to determine the next point $x_m$ for function evaluation; 
4. evaluate the function at $x_m$ and accordingly update $M$; 
5. repeat steps ii to iv until a termination predicate is satisfied.

A prominent acquisition function is based on the concept Expected Improvement (EI) [59], which pursues the idea of selecting the next point to sample based on the gain that is expected with respect to the currently known optimal configuration. More formally, considering without loss of generality a minimization problem, let $D_e$ the set of evaluation points collected so far, $D_u$ the set of possible points to evaluate in $D$ and $x_{min} = \arg \min_{x \in D_u} f(x)$; then the positive improvement function $I$ over $f(x_{min})$ associated with sampling a point $x$ is $I_{x_{min}}(x) = \max \{ f(x_{min}) - f(x), 0 \}$. Of course, since $f$ has not been evaluated on $x$, $I(x)$ is not known a priori; however, thanks to the predictive model $M$ fitted over past observations, it is possible to obtain the expected value for the positive improvement

$$EI_{y(x_{min})}(x) = E[I_{y(x_{min})}(x)] = \int_{-\infty}^{y(x_{min})} (f_{x_{min}} - c)p_M(c|x)dc$$

where $p_M(c|x)$ is the probability density function that the model $M$ associates to possible outcomes of the evaluation of $f$ at point $x$ [59]. High EI values are associated either with points that are regarded by the model as likely to be the minimum (high predicted mean) or with points whose corresponding value of the target function the model is uncertain about (high predicted variance). By selecting as next point for function evaluation the one that maximizes the EI, SMBO naturally balances exploitation and exploration: on one side it exploits model’s confidence to sample the function at points that are supposedly good candidates to be the minimum; on the other, it explores zones of the search space for which the model is uncertain, so as to increase its predictive power by iteratively narrowing uncertainty zones.
Chapter 3

Evaluating the relevance and complexity of TM tuning

The previous chapter has presented a vast array of possibilities to choose from when designing a TM library, e.g., STM vs HTM, lazy vs eager validation, word-based vs object-based granularity. Also, it has shown that not only the design is important, but the tuning of a TM at runtime can have a major impact in the overall performance of a system [91, 39, 47, 80, 81].

There are several parameters that can be tuned, and the trade-offs/implications on performance of tuning these knobs are non-trivial.

As so, in this section I am going to present an extensive experimental evaluation conducted as a sensitivity analysis on the impact of tuning different configuration parameters of TMs, namely: contention manager, number of threads, and TM backend.

The evaluation was performed using the following TMs: TinySTM, NOrec, SwissTM, TL2, and Intel TSX.

Table 3.1 shows the machines used for this study.

As for the benchmarks, this study covers a large portion of the benchmarks used in previous TM literature:

1. STAMP [13] - standard suite of benchmarks for TM that encompasses 8 different realistic applications, which generate very heterogeneous workloads.

2. STMB7 [52] - Based on OO7 [14] with many heterogeneous transactions over a large and complex graph of objects.
### CHAPTER 3. EVALUATING THE RELEVANCE AND COMPLEXITY OF TM TUNING

<table>
<thead>
<tr>
<th>Machine ID</th>
<th>Processor / Number of cores / RAM</th>
<th>HTM</th>
<th>RAPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine A</td>
<td>1 Intel Haswell Xeon E3-1275 3.5GHz / 4 (8 hyper-threads) / 32 GB</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Machine B</td>
<td>4 AMD Opteron 6172 2.1 Ghz / 48 / 32 GB</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3.1: Machines used in the TM parameters study.

3. Data structures - concurrent data structures, such as linked list, redblack tree, skip list, which are parallelized using transactions and frequently adopted in the TM literature[41]. Despite their simple and synthetic nature, these microbenchmarks have the advantage of generating easily predictable workloads that allow for stressing, in a controlled way, different building blocks of the TM system.

Note that, that this study evaluates both performance and energy efficiency (expressed as Joule/transaction).

The efficiency of existing TM implementations is strongly dependent on the workloads they face. Performance can be affected by a number of factors, including program inputs [42], phases of program execution [36], tuning of the internal parameters of the TM algorithms [40], as well as architectural aspects of the underlying hardware [17].

Fig. 3.1 conveys experimental evidence of the strong sensitivity of TM to the workload characteristics. The energy efficiency (in Fig. 3.1a) and throughput (in Fig. 3.1b) of various TMs are reported in different architectures and benchmarks. The data is normalized with respect to the best performing configuration for the considered workload. Fig. 3.1 shows that, in two different architectures and metrics, the optimal TM configuration differs significantly for each workload. Furthermore, choosing wrong configurations can cripple performance by several orders of magnitude. Interestingly, some TMs used in these experiments were designed to tackle various workloads [46, 43], but configuring them properly is non-trivial and they still cannot perform well for all workloads.

The presented data clearly serves as motivation for the study, which has a twofold purpose:

1. highlighting the relevance of the tuning of these parameters, by quantifying the impact that their tying has on the system's performance.

2. questioning whether it is possible to reduce the dimensionality of the self-tuning problem by avoiding to consider a subset of the parameter space considered in this study.
3.1 Contention Manager

Contention Managers have been a subject of study for several years. There are many relevant studies that advocate their use in TM [27, 12, 54, 83, 50], therefore, it was expectable to include CM as a possible dimension when building a TM system.

To infer the benefit of including CMs in TM, the study was conducted in two logical parts:

1. Evaluate how the different CM strategies compare against each other in the considered set of workloads.
2. Quantify the benefits achievable by including this parameter in the set of parameters to be automatically tuned.

Throughout the years many CM strategies were presented in literature [27, 12, 54, 83]. For this study, five possible strategies were considered:

1. Suicide - a thread aborts itself upon conflict detection.
2. Aggressive - a thread aborts the other one upon conflict detection.
3. Polite - a thread aborts itself and backoffs exponentially when a conflict is detected.
4. Karma - when a conflict is detected, the thread checks the amount of work performed and compares with the work of the other thread. In case it has done more work, it aborts the other, otherwise it aborts itself.
5. Timestamp - when a conflict is detected, the thread checks when it has begun the transaction and compares with the timestamp of the other thread. In case it started sooner, it aborts the other, otherwise it aborts itself.
The first three CM techniques, i.e., Suicide, Aggressive, and Polite are the most basic ones, since they do not require any information to take a decision. The other two are more elaborated ones, and rely on profiled information to perform informed decisions.

Performing this study required the integration of all these CMs into the considered TM backends. However, not all the considered TMs are amenable to be augmented with all the considered CMs. In particular, HTM and NOrec posed a major challenge, since the introduction of CM would require structural modifications at the basis of their design, which will decharacterize the TMs.

HTM limited support for CM is related to the fact that HTM does not employ instrumentation. NOrec, in the other hand, is optimized for low contention, using a lazy conflict detection, and invisible updates. Adapting NOrec to support aggressive policies counters the purpose of its design, since aborting another thread implies that the owner of the lock is known, which does not hold true for NOrec.

Due to the need for core modifications in both, they were left compatible with only Suicide and Polite, i.e., passive strategies.

Note that for this reason, both TM backends were omitted in the first part of this study, i.e., comparing the different TM approaches, in order to ensure the fairness of the comparison of of the CM strategies.

Before analyzing the results, it is also important to clarify that the implemented CM suicide is not the same as having the TM running without CM. For example, SwissTM uses a timestamp CM policy by default, moreover, TL2 relies on a backoff strategy at its backbone.

In order to assess the impact of ALL the CMs, this study has been performed only using TinySTM, SwissTM, and TL2, namely the backends that support all the aforementioned CM variants.

Fig. 3.2 reports the winning percentage of each CM on the picked workloads. A CM is considered to win in a given workload if the best throughput achieved on that workload used that CM strategy, assuming an optimal tuning along the other considered dimensions (so remember to recap which are the dimensions that you consider, at the beginning of the study): e.g., if TinySTM with four threads using Timestamp as CM has the best throughput for a given workload, then Timestamp wins on that workload. This plot shows a dominance of Timestamp, which wins around 38% of the times. All the other CMs are balanced, and the one that wins less is Karma with a 12% win rate. The overall equilibrium between the different strategies points towards the idea that all CM algorithms are important, in the sense that picking the best strategy highly depends on the workload's kind (the heterogeneity of the test suite allowed this demonstration).
In the search for confirming that each CM algorithm matters, i.e., the best CM is highly workload-dependent, we now look into this problem from a different angle. From the previous plot it is reasonable to infer that each CM could be important, since each has a significant winning percentage, still, the difference between each CM in terms of performance is missing, i.e., by how much a TM’s performance/energy efficiency can be further optimized by considering the CM configuration in the parameter space of the self-tuning problem. To this end, we compute the cumulative density function (CDF) of the distance from optimal (DFO) achievable by selecting a given CM manager, and optimizing the remaining three configuration parameters (repeat the other 3).

Fig. 3.3 corresponds to the obtained CDF after applying the described procedure. This complementary plot shows two important things. First it is possible to see that if the wrong CM is picked (assuming all the other parameters are properly tuned for the CM, i.e., number of threads and TM backend is the best configuration for the CM in the given workload) the worst possible scenario is to be 14% far from optimal. Second, this plot shows that the difference between each CM is usually not substantial, with Timestamp dominating in 70% of the sampling but for less than around 2%. This result suggests that the difference between each CM does not seem significant for its inclusion, considering the complexity that it is required to include CM logic and tuning it effectively. The auxiliary table 3.2 reinforces the latter conclusion. The difference between the highest average dist. from optimal and the worst, is 0.004, meaning that the worst CM is on average 0.4% further from optimal than the best, which is negligible. The median and the 90th percentile also points towards a minimum difference.

Not being able to verify the latter, and led by the motivation that the worst possible scenario can be 14% far from what it is possible to get, let us assume that in fact all CMs can be worth having.
CHAPTER 3. EVALUATING THE RELEVANCE AND COMPLEXITY OF TM TUNING

Figure 3.3: CDF of DFO for throughput.

![Figure 3.3: CDF of DFO for throughput.](image)

The study conducted on CM also evaluates how much can the inclusion of CM improve the original system. To address this, the approach was the following: for each workload the highest throughput with CM and without CM were picked, and the distance from optimal computed; using as optimal the highest value among them. Therefore, if the throughput is higher when using CM, then CM has a distance from optimal of 0 in that workload, and without CM has a distance from optimal that corresponds to how far it is from the latter.

Fig.3.4 illustrates the difference between having or not the CM logic integrated in the system. This plot corresponds to the CDF obtained by applying the previous rationale to compute the distance from optimal. It is possible to conclude that CMs generally have a positive impact on performance, but the gains are not that significant (50% of the population below 2%). Maybe surprisingly, the plot also reveals that the no-CM version sometimes wins against the CM one.

<table>
<thead>
<tr>
<th>Dist. from Optimal</th>
<th>Aggressive</th>
<th>Karma</th>
<th>Polite</th>
<th>Suicide</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0.012</td>
<td>0.015</td>
<td>0.016</td>
<td>0.012</td>
<td>0.011</td>
</tr>
<tr>
<td>Median</td>
<td>0.007</td>
<td>0.008</td>
<td>0.009</td>
<td>0.007</td>
<td>0.003</td>
</tr>
<tr>
<td>90th Perc.</td>
<td>0.022</td>
<td>0.031</td>
<td>0.041</td>
<td>0.029</td>
<td>0.031</td>
</tr>
</tbody>
</table>

Table 3.2: DFO avg., median, and 90th percentile for the CM variants.
This can be imputable to overhead in cases where implementing CM (at optimal configurations of other parameters) simply has no effect. In this sense, it seems to not compensate having this extra logic, given the additional cost in complexity in tuning it. Also, important to draw from this plot is that when using CMs it can go as worse as 12%.

Summarizing, the study highlighted a key result, which hold at least for the considered workloads and architecture: if one can optimize the degree of concurrency and the TM implementation, then the additional gains achievable by using the optimal CM scheme are negligible, being on average close to 2% and 4% at the 90-th percentile. Most importantly, the difference in throughput between them (even when one wins) is not substantial. Driven by the observation that there are cases where the difference is approx. 14% when picking the wrong CM, the decision to avoid discarding seemed reasonable.

Note that, the additional logic necessary for the inclusion of CM in the system will increase its overall complexity, and in particular the complexity of the self-tuning problem.

This result leads us to the decision of not considering further CM in the remainder of this sensitivity study, as well as among the parameters dynamically adjusted by ProteusTM.
3.2 Number of threads

The degree of parallelism (also called multi programming level in the database literature) have always played a major factor in performance since the introduction of parallel computing. As so, a large body of literature has advocated the need to self-tune the number of threads.

However, all those studies do not consider this problem into a multi-dimensional setting, i.e., how important is to tune threads when it is also possible to optimize other parameters.

The purpose of this study is therefore to demonstrate that if the number of threads is fixed, the performance in certain situations can be poor compared to what it is possible to achieve by self-tuning the parallelism degree. It should be referred that the main causes that could degrade throughput with a fixed number of threads are: length of the transactions, and the shared regions of memory accessed within or out of transaction context. Such is strictly workload-dependent.

In order to infer the worthiness of tuning the thread counts, the first approach is to verify how many times the different configurations for the parallelism degree result optimal, i.e., the highest throughput for a given workload was achieved using that number of threads.

Recall that, from now on, any contention management is used in the light of the study conducted in Section 3.1.

Fig.3.5 points towards the need for tuning the number of threads. More specifically it is possible to see that 48 threads seem to dominate on par with 32 threads. To address this problem deeper, another perspective will be taken on the problem.

The next experimentation consisted in picking for each workload the highest throughput achieved
with each fixed number of threads, e.g., for a red black tree workload TinySTM 4 threads, and SwissTM 8 threads are picked as best for their respective number of threads. After picking the best then it is calculated how far it is from optimal, using as optimal the highest throughput achieved in that workload when tuning along all the considering dimensions (i.e., TM backend and retry policy for HTM).

Fig.3.6 presents the CDF of the results employing the previous approach. The most interesting result is that the fixed thread configuration that results to be the best more often can also incur a performance degradation in the order of 80% w.r.t. the optimal achievable. The high sensitivity of this parameter is clearly illustrated in this plot, taking into account that also other configurations represent the optimal choice in a good percentage of the cases, and still can deliver performance as distant from the optimal as by 80%. The tuning of this parameter is then something to take into full consideration, and the broad domain of values that it can take make it undoubtedly hard to properly tune.

Table summarizes the main results that can be extrapolated from the CDF.

The conclusion that can be drawn from this study is that correctly identifying the number of threads is important. In fact, the study clearly showed that, even if one properly selects the TM implementation, the wrong choice of the degree of parallelism can still have a dramatic impact on
A higher number of threads can indeed achieve high levels of throughput, but when applied in improper workloads it can really damage the system’s performance (e.g., when they induce too much contention on data). As expected, the number of threads is a parameter that must be included in a system that aims to deliver a consistent top-notch performance across all kinds of TM workloads. Still, its tuning is complex, taking into account the parameter’s broad domain, and keeping in mind that an incorrect tune will decrease performance, as previously highlighted.

3.3 TM backend

This section investigates the impact that properly choosing the TM backend has on performance.

Previous work has shown that the choice of the best TM depends on the workload [42, 92]. Those studies, however, do not consider the impact of TM choice in a multi-dimensional tuning setting, i.e., when it is also possible to tune the TM along other dimensions.

This study aims at filling this gap, by proposing a study similar to what already done in previous sections for CM, and Number Of Threads studies.

The question for this study is whether or not should a TM library include multiple TM backends.

The TM backends considered were the following: TinySTM[47], NOrec[27], TL2[35], Intel TSX[95], and SwissTM[44].

As done for the previous parameters, first of all, this problem will be addressed by verifying the winrate of each TM backend, in order to investigate whether there is a single superior alternative.

Fig.3.7 depicts the winning percentage of each TM. For the considered set of workloads, we can see that HTM outperforms other TM implementations more than 60% of the times. Among the 4

<table>
<thead>
<tr>
<th># Threads</th>
<th>Distance from Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;25%</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>16</td>
<td>66</td>
</tr>
<tr>
<td>32</td>
<td>75</td>
</tr>
<tr>
<td>48</td>
<td>72</td>
</tr>
</tbody>
</table>

Table 3.3: Percentage of sampling with DFO lower than 25%, 50%, and 75%, for each number of threads.
remaining STM implementations, TL2 results the most efficient only in one workload. This result may suggest to remove TL2 from the set of alternative implementations supported by a self-tuning platform. Nonetheless, TL2 is still considered a state of the art TM and used as baseline in many academic papers [44, 47, 48], we decided not to exclude as possible backend for ProteusTM’s implementation.

Let us look at this problem from another perspective. Fig.3.8 further analyses the efficiency of the considered TM backends, in terms of delivered distance from optimal throughput. As expected, HTM is optimal in around 60% of the cases, but, as shown in the plot, it can be suboptimal by a 2x factor in 22% of the scenarios and in 10% of the considered workloads it is more than 70% slower than optimum. If the system would only rely on HTM, it would be performing well below its potential. Regarding the other TMs, SwissTM and TinySTM seem to be the second and third best, respectively. But, in some cases the performance for both can be completely far from optimal, approaching the 100% limit. As expected, this plot leaves the idea that the inclusion of a vast set of TM backends is desirable.

As usual, in Table 3.4 we provide a tabular representation of the CDF in Figure 3.8.
CHAPTER 3. EVALUATING THE RELEVANCE AND COMPLEXITY OF TM TUNING

Summing up, the conclusion that can be drawn is that there is no one size fits all TM backend, even in multi-dimensional self-tuning settings. i.e., tuning the TM backend is mandatory if a system wants to deliver optimal performance across the whole spectrum of possible workloads.

Despite HTM dominance in terms of winning percentage, the study revealed that its performance can be poor in a significant amount of cases, as depicted by the curve. Taking into account that each TM seems to excel with different workloads, the decision is to include them all in the final system’s design.

As it was shown, the correct tuning of the TM backend is a must, and always picking the best for a given workload is with no doubt a hard task to accomplish. The correct tuning is then of paramount importance, because switching to an improper TM can cripple performance to as bad as close to 100% far from the optimal.
Chapter 4

Building a Multi-dimensional TM

The previous chapter has highlighted the importance of properly tuning a number of parameters to guarantee optimal performance across heterogeneous workloads. Without employing runtime-adaptation, the possibility to build a solution that can deliver stable levels of performance across all scenarios seems unlikely. In essence, the high heterogeneity that ranges from different hardware characteristics, to different software workloads, poses a major challenge to the development of a simple, elegant TM that can ensure high efficiency when faced with highly heterogeneous workloads.

With these considerations in mind, in this Chapter I introduce ProteusTM, which allows developers to still enjoy the simplicity and ease of usage of the TM abstraction, while sparing them from the burden of tuning TM implementations to specific workloads.

Under the simple and elegant interface of TM, ProteusTM hides a large library of TM implementations. At run-time, ProteusTM relies on an innovative combination of learning techniques to pursue optimal efficiency via multi-dimensional adaptation of the TM implementation and its parameters.

At the heart of ProteusTM lie two key components:

- **PolyTM** is a polymorphic TM library that encapsulates state-of-the-art results from research in TM, and has the unique ability to transparently and dynamically adapt across multiple dimensions: (i) switch between different TM algorithms; (ii) reconfigure the internal parameters of a TM; (iii) adapt the number of threads concurrently generating transactions.

- **RecTM** is in charge of determining the optimal TM configuration for an application. Its basic idea is to cast the problem of identifying such best configuration as a recommendation problem [74]. This allows RecTM to inherit two highly desirable properties of state of the art Recommender
CHAPTER 4. BUILDING A MULTI-DIMENSIONAL TM

System (RS) algorithms: the ability to operate with very sparse training data, and to require only the monitoring of the Key Performance Indicator (KPI) to be optimized. This avoids intrusive instrumentation [79] and (possibly inaccurate) static code analysis [92] employed by other machine learning-based solutions.

In essence, ProteusTM applies Collaborative Filtering (CF) to the problem of identifying the best TM configuration that maximizes a user-defined Key Performance Indicator (KPI): e.g., throughput or consumed energy. ProteusTM aims to maximize the efficiency of TM applications by orchestrating a number of TM algorithms and the dynamic reconfiguration of their parameters.

The remainder of this Chapter is structured as follows. In Section 4.1 I introduce the high-level architecture of Proteus TM. In Section 4.2, a detailed presentation of PolyTM is provided. Section 4.3 consists in a highly detailed description of Recommender. Lastly, Section 4.4 presents the related work.

4.1 ProteusTM: High-level architecture

Fig. 4.9 provides an architectural overview of ProteusTM, and of its main internal components. In the following we briefly overview each of them and provide references to the corresponding sections of this chapter in which we provide their detailed presentation.

- **PolyTM**: consists of a Polymorphic TM library comprising various TM implementations. It allows for switching among TMs and reconfigure several of their internal parameters. It exposes transactional operators via an implementation of the standard TM ABI [70] (supported by GCC [58]). Details on PolyTM are provided in Section 4.2.

- **RecTM** is responsible for identifying the best configuration for PolyTM depending on the current workload. Details on it are provided in Section 4.3. RecTM is composed, on its turn, by the following sub-modules:

  1. **Recommender**: a RS that acts as a performance predictor and supports different CF algorithms. It receives the KPIs of explored configurations from the Controller, and returns ratings (i.e., predicted KPIs) for unexplored ones. Details on the Recommender are provided in 4.3.1.

  2. **Controller**: selects the configurations to be used and triggers their adaptation in PolyTM. It queries the Recommender with the KPI values from the Monitor, obtaining estimates for the ratings of unexplored TM configurations. Details on Controller are provided in 4.3.2.

  3. **Monitor**: this module collects the target KPI to (i) give feedback to the Controller about the
4.2 PolyTM: a Polymorphic TM Library

The PolyTM library encompasses a wide variety of TM implementations. It interacts with compilers, like GCC, via the standard TM ABI [58]. Each atomic block, written by the programmer using standard C/C++ constructs [70], is compiled into calls to the various modules of ProteusTM.

For every atomic block, GCC inserts a call to \texttt{tm_begin} and \texttt{tm_end}, which is direct to PolyTM. Also, two code paths are generated: a non-instrumented path, and a second one in which reads and writes to memory are instrumented with calls to PolyTM. The latter allows the code to arbitrate reads and writes, besides the begin and commit of transactions.

Behind the TM ABI interface, it is implemented in PolyTM several TM algorithms, and run-time support to switch among them: 4 STMs [26, 34, 46, 43], 2 HybridTMs [24, 65], and 2 HTMs [95, 1]. The system takes advantage of the dual compilation paths and use the instrumented one for the STMs. In contrast, HTMs — which automatically transactionalize reads and writes — execute the non-instrumented one. As it will be shown in Section 5.2, the dual path optimization is crucial to minimize overhead.

The compiled code is also instrumented to profile performance metrics in a lightweight and transparent manner. In particular, PolyTM collects the commits and aborts at each thread, and the energy consumed by the system. It also uses a dedicated adapter thread to change the TM configuration.

In the following, the mechanisms used by PolyTM to support run-time configuration changes are...
4.2.1 Switching Between TM Algorithms

Since the developed library must interact with the compiler via a single ABI, the different TM implementations are hidden under a common interface defined in PolyTM. Then, each thread uses a set of function pointers to this interface to process transaction operations. To switch between TMs, a thread switches the function pointers to a different implementation.

Running concurrent transactions with different TMs is not safe in general [92, 62]. So, PolyTM enforces an invariant: a thread may run a transaction in mode TM_A only if no other thread is executing a transaction in mode TM_B. The problem is illustrated in Fig. 4.10: at time $e_1$ the adapter thread tries to change the TM mode; if thread 2 immediately applied the change, it could run mode TM_B concurrently with thread 1 in TM_A. The above invariant guarantees correctness by forcing thread 2 to wait until $e_2$ to change to TM_B.

The invariant is enforced via an implementation based on the following three steps: (i) adapt parallelism degree (i.e., number of threads) from its current value, say $P$, to 0; (ii) change TM back-end; (iii) adapt parallelism degree back to $P$.

4.2.2 Adapting the Parallelism Degree

The maximum number of active threads is adapted using the synchronization scheme described in Algorithm 1.

Each application thread synchronizes with the adapter thread via a (padded) state variable. When executing a transaction for the first time, a thread is registered in PolyTM. This is simplified in the algorithm by assuming a maximum number of threads, although PolyTM supports an arbitrary number.

Upon starting a transaction, a thread $t$ sets the lowest bit in its state variable (line 10), whereas the
Algorithm 1 Changing the parallelism degree in PolyTM.

```c
1: const int RUN ← 1, BLOCK ← 1 << 32
2: padded var int threadState[MAX_THREADS] ← { 0 }
3: function disable-thread(int t) ▷ adapter thread
4:   int val ← fetch-and-add(threadState[t], BLOCK)
5:   while (val & RUN) val ← threadState[t]
6: function enable-thread(int t) ▷ adapter thread
7:   threadState[t] ← RUN
8:   signal(t) ▷ wakes up thread t (locking omitted)
9: function tm-start(int t) ▷ application thread
10: int val ← fetch-and-add(threadState[t], RUN)
11: if (val & BLOCK)
12:   fetch-and-sub(threadState[t], RUN)
13: cond-wait(t) ▷ checks it is still blocked after locking
14: ▷ ...omitting logic for tm-start...
15: function tm-end(int t) ▷ application thread
16: ▷ ...omitting logic for tm-end...
17: fetch-and-sub(threadState[t], RUN)
```

adapter thread sets the highest bit of $t$’s state variable when it wants to disable $t$ (line 4). These writes are performed atomically together with returning the state of $t$. Then, both adapter thread and $t$ can reason on who wins (a potential race): if $t$ sees only the lowest bit set, it is allowed to proceed and executes the transaction; otherwise, it must wait for the adapter to change the mode (line 13). The adapter inversely checks that only the highest bit is set, or else waits for $t$ to unset the lowest bit (line 5) — because $t$ was already executing a transaction.

These atomic operations are implemented with the primitives fetch-and-$op$ (e.g., $op = add$ would be `XADD` in x86). These primitives always succeed, and are cheaper than the traditional compare-and-swap loop [68, 30]. Furthermore, in the common case of the employed algorithm — a thread starting a transaction is not concurrently disabled — each thread performs the atomic operation on a variable residing (with high probability) in its cache and without contention. In this case, the latencies (in processor cycles), in Machine A are 17 cycles for a fetch-and-add and 32 for a compare-and-swap. As such, the cost for managing the number of active threads is quite limited, for instance when compared to the begin and commit of a hardware transaction (>120 cycles [76]).

Also, a conditional variable is used, and associated with each thread $t$, for $t$ to wait on, in the case it is disabled. The details of its management have been omitted, for simplicity of presentation.

PolyTM guarantees that a reconfiguration always terminates: a thread eventually commits a pending transaction, or else aborts and checks whether it was disabled — assuming finite atomic blocks. Hence, the duration of a reconfiguration depends on the longest running transaction. This, however, does not impair the efficiency of PolyTM’s reconfiguration: in-memory transactions
are generally very fast (given that they do not entail I/O) [89, 61].

In addition, the success of a reconfiguration does not rely on threads to eventually call into ProteusTM. This is crucial to cope with applications whose threads may wait for events (e.g., client requests) and do not run atomic blocks often.

Note that, depending on the application, it may not be safe for PolyTM to permanently disable an arbitrary thread: for instance, a web server may have a single thread accepting requests. To account for such cases, in which it is impossible to know the application’s semantics, a library call is provided for the programmer to forbid PolyTM from disabling a specific thread (e.g., to tune the parallelism degree). Such a thread, however, may be disabled temporarily to allow switching the TM algorithm, which is a brief procedure as noted above.

### 4.2.3 Adapting HTM internals

PolyTM’s optimization encompasses other configuration parameters related to contention management [51]. Specifically, PolyTM integrates a scheme for HTM [40] that is affected by two parameters: (i) the budget of retries using HTM for a transaction, (ii) whether, upon a capacity abort, the budget should be decreased by one, halved, or fully consumed.

In fact, different contention management policies can co-exist without threatening correctness [51]. Hence, both parameters can be changed at any point without synchronization.

### 4.3 RecTM: a Recommender System for TM

RecTM optimizes PolyTM via a black-box methodology that relies on a novel combination of off-line and on-line learning. In short, it operates according to the work-flow of Algorithm 2:

(i) build a training set by profiling the KPI of an initial set of applications in the encompassed TM configurations (line 1);
(ii) instantiate a CF-based performance predictor based on the training set obtained off-line in (i) (lines 2 and 3);
(iii) upon deploying a new application or detecting a change of the workload, profile on-line the application over a small set of explored configurations (lines 4 and 5);
(iv) recommend a configuration for the workload (line 6).

In the following, the building blocks of RecTM will be detailed.
4.3. RECOMMENDER SYSTEM FOR TM

Algorithm 2 RecTM work-flow

1: Off-line performance profiling of an initial training set of applications.
2: Rating distillation and construction of the Utility Matrix (Section 4.3.1).
3: Selection of CF algorithm and setting of its hyper-parameters (Section 4.3.1).
4: Upon the arrival of a new workload (Section 4.3.3):
5: Sample the workload on a small set of initial configurations (Section 4.3.2).
6: Recommend the optimal configuration (Section 4.3.1).

4.3.1 Recommender: Using Collaborative Filtering

RecTM casts the identification of the optimal TM configuration for a workload into a recommendation problem, which it tackles using Collaborative Filtering (CF), an efficient and simple technique for rating prediction [86].

A key challenge to successfully apply CF in predicting the performance of TM applications, is that CF assumes the ratings in a predetermined scale (e.g., a preference from 0 to 10). The absolute KPI values produced by different TM applications, instead, can span orders of magnitude (e.g., from millions [13] to few txs/sec [52]). Further, KPI values of specific configurations provide no indication on the max/min KPI that the application can obtain, impairing their normalization.

Recommender tackles this issue with an innovative technique, which let us address it as rating distillation. This function maps KPI values of diverse TM applications onto a rating scale that can be fruitfully exploited by CF to identify correlations among the performance trends of heterogeneous applications.

The Rating Heterogeneity Problem. Ratings are stored in a Utility Matrix (UM) \( A \), of which each row \( u \) represents a workload and each column \( i \) is a TM configuration: \( A_{u,i} \) is the rating of configuration \( i \) for workload \( u \) (i.e., in the chosen domain, it expresses the performance of \( i \) in \( u \) for a given KPI metric). To illustrate the problem, let us populate the UM directly with sampled KPI values (e.g., throughput): \( \begin{pmatrix} 1 & 2 & 3 \\ 30 & 20 & 10 \\ 100 & 200 & ? \end{pmatrix} \), which contains information on applications \( A_1 \) and \( A_2 \) profiled with configurations \( C_1 \), \( C_2 \) and \( C_3 \) and \( A_3 \) profiled only at \( C_1 \) and \( C_2 \). Let us assume that \( C_i \) is an application running with a given TM and \( i \) threads. From the matrix, it is possible to infer that \( A_1 \) can scale, as its performance increases linearly with the number of threads; \( A_2 \) does not, since its performance, though higher in absolute value than \( A_1 \)'s, decreases as the number of threads grows. Assume that it is required to predict the rating for \( A_{4,3} \). Note that \( A_3 \) exhibits the same linear trends of \( A_1 \): for this reason, a likely value for \( A_{3,3} \) would be 300. Next, it will be shown why well-known CF techniques can be misled because of the heterogeneity of the ratings' scales in the UM.

The Need for Normalization. The most common similarity functions in KNN CF are the Euclidean, Cosine and Pearson [74]. The first cannot be applied to heterogeneous ratings, because
it is based on the scale-sensitive Euclidean distance: in the example above, it would incorrectly regard \( C_2 \) as more similar to \( C_3 \) than \( C_1 \). The other two are scale-insensitive, so they are able to identify \( C_1 \) as similar to \( C_3 \). However, they would yield an incorrect prediction in absolute value, as it will lie on \( C_1 \)'s scale, which is different from \( C_3 \)'s.

A similar shortcoming applies to MF CF. The \( P \) and \( Q \) matrices — recall Section 2.3 — are typically obtained by means of Stochastic Gradient Descent [74]: starting from random matrices, this technique iteratively tries to minimize the fitting error of \( P^TQ \) over \( A \). Thus, it is prone to over-fitting around the highest absolute value ratings, yielding poor overall accuracy.

A solution to these problems is to normalize the entries in UM. An effective normalization function should fulfill two requirements: (i) to transform entries in the UM so that similarities among heterogeneous applications can be mined and (ii) to enable the application of conventional CF techniques.

Note that feature normalization is often performed in Machine Learning (ML): the most notable example is in Artificial Neural Networks, which normalize input features in the range \([0,1]\) [6]. In ML, however, normalization is performed on the input features, whose values are fully known for samples in the training set and for queries. In contrast, in ProteusTM, the normalization has to be performed on the UM, which contains values corresponding to the output feature KPI, and whose entries are not all known. Next, it is described how ProteusTM normalizes ratings to meet the two aforementioned requirements and, thus, enables CF to optimize TM applications.

**Normalization in the Recommender.** If the minimum and maximum KPIs of an application were known a priori, they could be mapped to a homogeneous rating scale with a simple, per workload, normalization. Since KPIs of applications can take arbitrary values, then this ideal solution cannot be used.

The rating distillation used by the Recommender approximates the ideal approach with a mapping function that, for any workload \( w \) in the UM, ensures: (i) the ratio between the performance of two configurations \( c_i, c_j \) is preserved in the rating space, i.e., \( \frac{k_{w,c_i}}{k_{w,c_j}} = \frac{r_{w,c_i}}{r_{w,c_j}} \); and (ii) the ratings of the corresponding configurations, \( r_{w,c_i} \), are distributed (assuming a maximization problem) in the range \([0,M_w]\), so as to minimize the index of dispersion of \( M_w \): \( D(M_w) = \frac{\text{var}(M_w)}{\text{mean}(M_w)} \).

Property (i) ensures that the information about the relative distances of two configurations is correctly encoded in the rating spaces. Property (ii) aligns the scales that express the ratings of each workload \( w \) to use similar upper bounds \( M_w \), which are tightly distributed around their mean value.

This function is defined in Algorithm 3. The rating of each row is obtained by normalizing its KPI with respect to a column \( C^* \in C_M \), so to minimize the index of dispersion among the resulting
Algorithm 3 Rating Distillation function in ProteusTM.

1: for $C_i \in C_1 \ldots C_M$ do  
2: Normalize Matrix $KPI$ w.r.t. $C_i$  
3: Collect the vector $M_w$ with the max values per row  
4: Compute $\text{mean}_i(M_w)$ and $\text{var}_i(M_w)$  
5: end for  
6: Return $\text{argmin}_{i \in 1 \ldots M} \frac{\text{var}_i(M_w)}{\text{mean}_i(M_w)}$

maximum ratings in the normalized domain.

Note that not only does this function reduce the numerical heterogeneity of ratings; it also projects all the elements of the matrix to a semantically common domain: now, a rating $k$ for configuration $i$ can be seen as "configuration $i$ delivers performance that are $k$ times the reference one". While an absolute throughput of 5K txs/sec may correspond to either a good or a bad performance depending on the application, the employed rating function gives ratings a "more universal" meaning. Also, minimizing the dispersion of the maximum values of the scales, allows for aligning the upper extreme of the rating distributions of each application (i.e., matrix row) to a common value: the tighter the distribution around a common value $M_w$, the closer it approximates an ideal "omniscient" normalization.

Tuning the Recommender. Mahout [72] is used in Recommender, a ML framework containing several CF algorithms. This design choice allows the Recommender to seamlessly leverage a vast library of CF techniques, rather than binding it to a single one.

The Recommender uses the training UM to choose one of the available CF algorithms, to adopt at run-time, and properly tunes its parameters (e.g., similarity function). Determining the best learning algorithm and its hyper-parameters, given a training set, is a challenge that falls beyond the domain of CF [3]. Recommender uses an approach based on random-search [4] and $n$-fold cross-validation [7, 87, 57].

4.3.2 Controller: Explorations Driven by Bayesian Models

The Controller uses Sequential Model-based Bayesian Optimization (SMBO) [57] to drive the on-line profiling of incoming workloads, to quickly identify optimal TM configurations.

SMBO is a strategy for optimizing an unknown function $f : D \rightarrow \mathbb{R}$, whose estimation can only be obtained through (possibly noisy) observation of sampled values. It operates as follows: (i) evaluate the target function $f$ at $n$ initial points $x_1 \ldots x_n$ and create a training set $S$ with the resulting $\langle x_i, f(x_i) \rangle$ pairs; (ii) fit a probabilistic model $M$ over $S$; (iii) use an acquisition function $a(M, S) \rightarrow D$ to determine the next point $x_m$; (iv) evaluate the function at $x_m$ and accordingly
Table 4.5: TM applications used in the evaluation. These 15 benchmarks span a wide variety of workloads and characteristics.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of Code</th>
<th>Atomic Blocks</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structures</td>
<td>3702</td>
<td>12</td>
<td>Concurrent Red-Black Tree, Skip-List, Linked-List and Hash-Map with workloads varying contention and update ratio.</td>
</tr>
<tr>
<td>STMBench7 [52]</td>
<td>8623</td>
<td>45</td>
<td>Based on OO7 [14] with many heterogeneous transactions over a large and complex graph of objects.</td>
</tr>
<tr>
<td>TPC-C [88]</td>
<td>6690</td>
<td>5</td>
<td>OLTP workload with in-memory storage adapted to use one atomic block encompassing each transaction.</td>
</tr>
<tr>
<td>Memcached [78]</td>
<td>12693</td>
<td>120</td>
<td>Caching service with many short transactions that are used to read and update the cache coherently.</td>
</tr>
</tbody>
</table>

update $M; v$) repeat steps (ii) to iv) until a stopping criterion is satisfied.

**Acquisition function.** Controller uses as acquisition function the criterion of Expected Improvement (EI) [59], which selects the next point to sample based on the gain that is expected with respect to the currently known optimal configuration. More formally, considering without loss of generality a minimization problem, let $D_e$ be the set of evaluation points collected so far, $D_u$ the set of possible points to evaluate in $D$ and $x_{min} = \arg\min_{x \in D_u} f(x)$. Then the positive improvement function $I$ over $f(x_{min})$ associated with sampling a point $x$ is $I_{x_{min}}(x) = \max\{f(x_{min} - f(x), 0\}$.

Since $f$ has not been evaluated on $x$, $I(x)$ is not known a priori; however, thanks to the predictive model $M$ fitted over past observations, it is possible to obtain the expected value for the positive improvement:

$$EI_{y(x_{min})}(x) = E[I_{y(x_{min})}(x)] = \int_{-\infty}^{y(x_{min})} (f(x_{min} - c)p_M(c|x)dc.$$  

Here, $p_M(c|x)$ is the probability density function that the model $M$ associates to possible outcomes of the evaluation of $f$ at point $x$ [59]. High EI values are associated either with points that are regarded by the model as likely to be the minimum (high predicted mean), or with points whose corresponding value of the target function the model is uncertain about (high predicted variance). By selecting as next point for evaluation the one that maximizes the EI, SMBO naturally balances exploitation and exploration: on one side it exploits model’s confidence to sample the function at points that are supposedly good candidates to be the minimum; on the other, it explores zones of the search space for which the model is uncertain, to increase its predictive power by iteratively narrowing uncertainty zones.

**Computing $p_M(c|x)$.** The Controller computes $p_M(c|x)$ with an ensemble of CF predictors, and obtains predictive mean $\mu_x$ and variance $\sigma^2_x$ of $p(c|x)$ as frequentist estimates over the output of its individual predictors evaluated at $x$. It then models $p_M(c|x)$ as a Gaussian distribution $\sim N(\mu_x, \sigma^2_x)$. Assuming a Normal distribution for $p(c|x)$ is frequently done in SMBO [57] and
4.3. RECTM: A RECOMMENDER SYSTEM FOR TM

<table>
<thead>
<tr>
<th>Machine ID</th>
<th>Processor / Number of cores / RAM</th>
<th>HTM</th>
<th>RAPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine A</td>
<td>Intel Haswell Xeon E3-1275 3.5GHz / 4 (8 hyper-threads) / 32 GB</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Machine B</td>
<td>AMD Opteron 6172 2.1 Ghz / 48 / 32 GB</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.6: Machines used in the experimental test-bed.

other optimization techniques [71] to ensure tractability. Given a Gaussian distribution for \( p_M(c|x) \), \( EI_y(x_{min}) = \sigma_x \left[ u \Phi(u) + \phi(u) \right] \), where \( u = \frac{y(x_{min}) - \mu_x}{\sigma_x} \) and \( \Phi \) and \( \phi \) represent, respectively, the probability density function and cumulative distribution function of a standard Normal distribution [59].

More in detail, the Controller builds a bagging ensemble [8] of \( k \) CF learners, each trained on a random subset of the training set. Then, it computes \( \mu_x \) as the average of the values output by the single predictors, and \( \sigma^2_x \) as their variance. In ProteusTM, a 10 bagged models is used; it is important to highlight that the cost of employing them instead of a single one is negligible, mainly because they are only queried during profiling phases.

**Stopping Criterion.** As discussed, SMBO requires the definition of a predicate to stop exploring new configurations.

Controller uses a stopping criterion that seeks a balance between exploration and exploitation by relying on the notion of EI: it uses the estimated likelihood that additional explorations may lead to better configurations. More precisely, the exploration is terminated after \( k \) steps when: (i) the EI decreased in the last 2 iterations; (ii) the EI for the \( k \)-th exploration was marginal, i.e., lower than \( \epsilon \) with respect to the current best sampled KPI; (iii) the relative performance improvement achieved in the \( k - 1 \)-th iteration did not exceed \( \epsilon \). In Section 5.3, it is evaluated the effectiveness of this policy.

4.3.3 Monitor: Lightweight Behavior Change Detection

The Monitor periodically gathers KPIs from PolyTM. These are used for two tasks: (i) while profiling a new workload, they are fed to the Controller, providing feedback about the quality of the current configuration; (ii) at steady-state, they are used to detect a workload change. The Monitor implements the Adaptive CUSUM algorithm to detect, in a lightweight and robust way, deviations of the current KPI from the mean value observed in recent time windows [2]. This
allows the Monitor to detect both abrupt and smooth changes and to promptly trigger a new profiling phase in Controller. Note that environmental changes (e.g., inter-process contention or VM migration) are indistinguishable from workload changes from the perspective of the implemented behavior change detection.

4.4 Related work

The work lies at the intersection of three major research fields: optimization of TM applications, performance prediction via RS, and experiment-driven optimization of computer systems.

Optimization of TM applications. The most researched problem in TM self-tuning is choosing the number of active threads. Proposed solutions rely on analytical modeling [82], off-line machine learning [79], or exploration-based strategies [36]. Wang et al. [92] use Artificial Neural Networks and programmer heuristics to determine the best TM for an application (excluding HTM). Workload characterization uses static analysis and runtime detailed profiling. Tuner [40] exploits hill climbing and reinforcement learning to adaptively determine the retry-on-abort policy for HTM.

These solutions optimize a single aspect of TM. Moreover, purely exploration-based solutions [36, 40] are impractical in high dimensional spaces, as the number of configurations to explore rapidly grows with the parameters to optimize. ProteusTM, instead, is effective in optimizing in a large space (we considered 130 configurations over 4 parameters). Thanks to the capability of CF techniques to deal with sparse information and to the model applied to steer online explorations, ProteusTM operates in even higher dimensional problems by simply including more configurations as extra columns of the UM.

In addition, all surveyed solutions need either preliminary code analysis or intrusive instrumentation for workload characterization. These add complexity to the code of TM algorithms and overhead to the application. ProteusTM, conversely, relies solely on profiling high-level KPIs, which incurs minimal overhead and maximizes portability. Note that ProteusTM’s work-flow is fully automated, avoiding the need for programmer heuristics [92]. Finally, ProteusTM avoids off-line training on the target application, unlike other ML approaches [79].

Performance prediction via Recommenders. To the best of my knowledge, Paragon [32], Quasar [33] and U-CHAMPION [73] are the only systems relying on RS for performance prediction, job scheduling and resource provisioning. They characterize an incoming job via random sampling of a fixed number of configurations and then apply MF-based CF.
ProteusTM differs from these works in three key aspects: (i) it relies on a novel rating distillation function that identifies similarity patterns among the performances of heterogeneous applications. One noteworthy finding of the developed work is that this pre-processing step, not used in previous works, is of paramount importance to achieve high accuracy in the TM domain; (ii) ProteusTM leverages model-based techniques to determine which and how many configurations to experiment with during the run-time sampling phase of a new workload: it was shown that it outperforms random sampling, yielding lower sampling time and higher accuracy; (iii) it integrates both MF- and KNN-based CF, being able to determine the best one to employ, depending on the training data.

Experiment-driven optimization. The classic approach in this field is dynamic sampling: a performance model of the system is initialized, by evaluating its performance corresponding to some randomly chosen configurations; then, the next experiment to run is chosen on the basis of the observations progressively collected [96, 45, 93, 71]. The proposal that is closest to the developed work is iTuned [45], which optimizes configuration parameters of a database for a target application. It exploits Gaussian processes [75] to build a performance model of the application and uses the EI acquisition function to determine the next experiment to run. However, iTuned does not use knowledge of previous optimized workloads: by leveraging CF, ProteusTM alleviates the need for the initial static sampling phase, and is able to perform accurate recommendations after a short on-line adaptive sampling phase.
Chapter 5

Evaluation

This section provides an extensive validation of ProteusTM. Section 5.1 introduces the testbed, applications, and accuracy metrics used. In Section 5.2 the overhead incurred by PolyTM to provide self-tuning capabilities is assessed. In Section 5.3, the effectiveness of RecTM's components is evaluated in separate. Finally, in Section 5.4 the ability of ProteusTM to perform online optimization of dynamic workloads is evaluated.

5.1 Experimental Test-Bed

ProteusTM was deployed in two machines with different characteristics (described in Table 4.6) and used a wide variety of TM applications (summarized in Table 4.5). Over 300 workloads were considered, which are representative of heterogeneous applications, from highly to poorly scalable, from HTM to STM friendly [42]. Moreover, three KPIs were tested: execution time, throughput and EDP (Energy Delay Product, a popular energy efficiency metric [56]). The energy consumption was measured via RAPL [29] (available on Machine A).

The system optimizes the KPI by tuning the four dimensions listed in Table 5.7. Overall, a total of 130 TM configurations are considered for Machine A and 32 for Machine B.

Evaluation metrics. The performance of ProteusTM is evaluated by considering 2 accuracy metrics: Mean Average Percentage Error (MAPE) and Mean Distance From Optimum (MDFO).

Noting \( r_{u,i} \) the real value of the target KPI for workload \( u \) when running with \( i \) as configuration, \( \tilde{r}_{u,i} \) the corresponding prediction of the Recommender, and \( S \) the set of testing \( (u,i) \) pairs, MAPE is defined as: 

\[
\text{MAPE} = \frac{\sum_{(u,i) \in S} |r_{u,i} - \tilde{r}_{u,i}|}{r_{u,i}}.
\]
### Table 5.7: Parameters tuned by ProteusTM. STMs are TinySTM [46], SwissTM [43], NORec [26] and TL2 [34].

<table>
<thead>
<tr>
<th>Machine ID</th>
<th>TM Backend</th>
<th># threads</th>
<th>HTM Abort Budget</th>
<th>HTM Capacity Abort Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine A</td>
<td>STMs and TSX [95]</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2.4,8,16,20</td>
<td>Set budget to 0; decrease budget by 1; halve budget</td>
</tr>
<tr>
<td>Machine B</td>
<td>STMs</td>
<td>1,2,4,6,8,16,32,48</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Table 5.8: Overhead (%) incurred by ProteusTM for different TM and # threads. Results are an average across ten runs.

<table>
<thead>
<tr>
<th>#threads</th>
<th>TL2</th>
<th>NOrec</th>
<th>Swiss</th>
<th>Tiny</th>
<th>HTM-opt</th>
<th>HTM-naive</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>&lt;1</td>
<td>1</td>
<td>&lt;1</td>
<td>3</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>&lt;1</td>
<td>&lt;1</td>
<td>&lt;1</td>
<td>4</td>
<td>8</td>
<td>19</td>
</tr>
</tbody>
</table>

Noting with $i_u^*$ the optimal configuration for workload $u$ and with $\hat{i}_u$ the best configuration identified by the Recommender, the MDFO for $u$ is computed as: $\sum_{(u, i) \in S} |r_{u,i} - r_{u,i^*}| / r_{u,i^*}$.

MAPE reflects how well the CF learner predicts performance for an application. In contrast, MDFO captures the quality of final recommendations output by the Recommender.

### 5.2 Overhead Analysis and Reconfiguration Latency

The overhead of PolyTM, is now assessed i.e., the inherent steady-state cost of supporting adaptation. Let us proceed to the performance comparison of a bare TM implementation $T$ with that achieved by PolyTM using $T$ without triggering adaptation.

Table 5.8 summarizes the results averaged across all benchmarks. The contention management for HTM is set to decrease linearly the retries starting from 5 (a common setting [95, 60]). It is also shown the overhead of the optimized code path, employed for HTM, and the one resulting from the default GCC instrumentation (fully instrumented path).

These experiments reveal overheads consistently $<5\%$ for STMs and only slightly larger for HTMs (at most $8\%$). The lower STM overhead is justifiable considering that STMs natively suffer from instrumentation costs that end up amortizing most of the additional overhead introduced by PolyTM.

Also, it is assessed the average latency of a typical reconfiguration in PolyTM to switch TM
5.3. QUALITY OF THE PREDICTION AND LEARNING PROCESSES

<table>
<thead>
<tr>
<th>Benchmark (Machine)</th>
<th># Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-C (Machine A)</td>
<td>21  91  213  3419  N/A  N/A</td>
</tr>
<tr>
<td>Memcached (Machine B)</td>
<td>2   8   28   145   1103  1849</td>
</tr>
</tbody>
</table>

Table 5.9: Reconfiguration (TM and #threads) latency (\(\mu\)sec).

algorithms (which also entails changing the number of threads). The results, shown in Table 5.9, encompass two heterogeneous workloads: Memcached uses 100× shorter transactions than TPC-C. The results highlight the practicality of the system’s reconfiguration algorithm. Even in the worst case of large transactions in TPC-C, the latency is negligible: in fact, this is only incurred during the exploration phase, which, as we shall see, is kept very short by ProteusTM.

5.3 Quality of the Prediction and Learning Processes

Let us evaluate each of RecTM’s components by means of a trace-driven simulation. Traces of real executions were collected using a subset of the test cases (namely, STAMP and Data Structures), averaging the results over 5 runs.

The data-set was split into a training set (30%) and a test set (70%). The training set is used to choose and tune the CF algorithm (Section 4.3.1) and to instantiate the predictive model. It is used 10 learners for the bagging ensemble, as this is a typical value [57, 87]. To simulate sampling the performance of the application in a given configuration, the corresponding value from the test set is inserted in the UM of the Recommender.

Rating distillation. The effectiveness of the distillation function was assessed versus several UM preprocessing techniques:

(i) **No normalization**: CF is applied on the UM containing raw KPI samples. This is equivalent to Quasar [33];

(ii) **Normalization w.r.t. max.**: entries in the UM are relative to the highest value, supposed to be known a priori. It resembles Paragon’s approach [32], where the machine’s peak instructions/sec rate is used as normalizing constant;

(iii) **Ideal normalization**: the scheme described in Section 4.3.1;

(iv) **Row-column subtraction**: noted \(RC\), is typically employed in CF to cope with biases in users and item ratings [74]. It consists in removing from each known rating the average value of the corresponding row; then, the average value per column — computed after the first subtraction — is subtracted;
CHAPTER 5. EVALUATION

Figure 5.11: Rating distillation for Exec. Time on Machine A.

(v) Rating distillation: used in ProteusTM (Algorithm 3).

A subset of results are shown, focusing on execution time KPI on Machine A employing KNN with cosine similarity. The number of randomly chosen known ratings per row is varied and then MAPE and DPO are accordingly computed.

Fig. 5.11 shows that using no normalization, or normalization w.r.t. the maximum performs very poorly, both in terms of MAPE (Fig. 5.11a) and MDFO (Fig. 5.11b). Both yield similar results as a normalization w.r.t. any constant. RC is subject to lower MAPE than the two aforementioned normalizations, yet its accuracy is significantly lower than rating distillation’s, both in terms of MAPE and MDFO. Also, the approach of ProteusTM closely follows the ideal normalization. To ensure a fair comparison, the same training set was used, and not forcing the presence of the column used for normalization among the profiled configurations for ProteusTM.

Controller. Let us now evaluate the proposed EI-based approach (called EI) with respect to a randomized sampling approach, used in Quasar and Paragon [32, 33], and two other SMBO approaches using acquisition functions different from EI: Variance explores configurations with high uncertainty for the underlying model (i.e., high variance/mean ratio); Greedy explores the configuration with highest predictive mean.

The simulation proceeds in rounds: each one profiles the target workload on the reference configuration chosen by the rating distillation function; then the sampling phase begins. Afterwards, the Recommender produces a recommendation for the optimal configuration, noted $\hat{c}^*$. If such a configuration is explored, then the optimization is concluded; otherwise, a final exploration of $\hat{c}^*$ is performed. The final recommendation $c^*$ is the one which, among those explored, yields the best performance. The MDFO is computed on the basis of $c^*$ and the MAPE is an average
5.3. QUALITY OF THE PREDICTION AND LEARNING PROCESSES

Figure 5.12: Controller’s exploration policies for EDP on Machine A (left figures) and Exec. Time on Machine B (right figures).

MAPEs computed per workload.

In Fig. 5.12a, it is reported the MDFO for EDP (on Machine A). The EI exploration policy is able to identify a high quality solution requiring, on average, less explorations than any competitor. Fig. 5.12b shows that the 80-th percentile of the DFO obtained by EI — after 5 explorations — is less than 10%. Note that, the EDP KPI was the most challenging to optimize: hence, the latter result represents a lower bound on the system’s accuracy.

In Fig. 5.12d, it is shown the MDFO when optimizing execution time (on Machine B): once again, the EI-based Controller’s exploration performs best. Fig. 5.12c shows the MAPE per explorations. Interestingly, the Variance policy has the best mean prediction accuracy. However, as it does not aim at sampling potential optimal solutions, but only at reducing uncertainty, it does not learn the behavior of the target function for potentially good configurations. Thus, the quality of the recommended configurations is significantly worse than EI’s (see Fig. 5.12d).

Finally, let us compare EI policy with random sampling in Figs. 5.12a and 5.12d: taking 5% distance as reference, EI achieves a number of explorations vs MDFO trade-off that is up to $4 \times$ better than its competitor. This highlights the effectiveness of the SMBO-based approach over
simpler sampling techniques used in recent systems [33, 32]).

**Stopping criterion.** Let us now evaluate the employed stopping heuristic (Section 4.3.2), called *Cautious* \(C\) in the plots). Cautious is compared with a *Naive* stopping rule \(N\) that blindly trusts the model, by stopping explorations when the expected improvement over the best known configuration falls \(< \epsilon\). The results are shown in Fig. 5.13, portraying the sensitivity of both heuristics to \(\epsilon\).

For any fixed \(\epsilon\), it is possible to observe that the Naive predicate chooses consistently a worse configuration than the one of the Cautious heuristic: blindly trusting the predictive model results in an excessively eager policy, which does not provide the model with enough training data to achieve adequate accuracy.

As expected, the plots also show that the lower \(\epsilon\), the lower the obtained MDFO. Notably, for \(\epsilon = 0.01\), the Controller achieves, in 90% of the cases, MDFO of only 5% when considering execution time on and 12% when optimizing EDP. This comes at the price of a higher number of explorations. Controller is able to keep this price very low, by requiring, on average, a similar number of explorations of a policy that performs a fixed amount of explorations and is tuned to deliver the same mean performance. This confirms the effectiveness of the Controller in wisely determining the duration of the profiling phase, by striking a balanced trade-off between the extent of on-line exploration and final performance.

**Comparison with ML approaches.** Let us now compare ProteusTM with an approach based on the same technique proposed by Wang et. al [92] to automate the choice of the TM algorithm for a given workload. This approach relies on workload characterization data to train a ML-based classifier that is used to predict the best TM configuration for a given workload. The workload characterization uses 17 features: e.g., duration of transactions, data access patterns, and level

![Figure 5.13: Comparing early-stop exploration predicates.](attachment:image.png)
5.3. QUALITY OF THE PREDICTION AND LEARNING PROCESSES

of data contention. Wang et al. also uses static analysis to obtain other features, e.g., the number of atomic blocks. This step was not performed but we extend the set of features proposed by the authors with information on contention management (Aggressive, Suicide, Polite, Karma, Timestamp). These features were not considered by the authors, but they were found to be highly correlated with performance.

The simulation for ProteusTM evolves as previously explained. For the ML competitor, instead, a workload is first profiled over a reference configuration (TinySTM, 4 threads) and then the ML is invoked to predict the best configuration. After it, the MDFO is computed for this predicted configuration.

300 STAMP and Data Structures workloads were used in Machine A, and splitted randomly into training and test sets: 30-70 and 70-30 train-test splits. For ProteusTM, the training set is the UM corresponding to the selected workloads; for ML approaches, the training set is composed, for each workload, by the aforementioned features and the identifier of the best configuration as target class. The target KPI is throughput.

3 ML algorithms, implemented in Weka [53] were considered: Decision Trees (CART), Support Vector Machines (SMO), and Artificial Neural Networks (MLP) [6]. Their parameters were chosen via random search optimization [4], which evaluated 100 combinations with cross-validation on the training set.

Fig. 5.14 reports the CDF of the DFO of each technique over 10 runs. The data shows the superiority of ProteusTM relatively to pure ML approaches. In particular, with 30% training set, ProteusTM already delivers a DFO of 1.6% against the 10% of the ML competitors, and a 90-th percentile of 3.5% against 25% of CART (the best alternative). Also, by increasing the training set to 70%, ProteusTM delivers a DFO of 1.3% and a 90-th percentile of 3%, against 6.8% DFO.

![Figure 5.14: Comparison vs Machine Learning based techniques.](image)
and 21% 90-th percentile of the best alternative (SMO).

Note that the DFO of ProteusTM is similar (both in mean and 90-th percentile) in both cases, whereas ML greatly benefits from more training data. This difference can be explained by the number of explorations required by ProteusTM to perform its profiling phase (with threshold $\epsilon = 0.01$): at 30% training, the 90-th percentile number of explorations is 7, but this lowers to 6 with 70% training set. This means that ProteusTM delivers high accuracy also in presence of scarce training data, by autonomously exploring more.

The evaluation suggests that detecting similarities on the KPI is more effective than statistically inferring relationships from training data. This possibly depends on two, tightly intertwined, causes: (i) thanks to the employed novel normalization, using CF is more robust than ML, as it is based on direct KPI observations, rather than on learning the mapping of input to output features; (ii) the adaptive profiling phase proved to be more effective than a one-shot classification-based solution.

Figure 5.15: Performance of four applications when their workload changes three times. It is shown the performance obtained with ProteusTM, and three additional fixed configurations, each one corresponding to an optimum in each workload.
5.4 Online Optimization of Dynamic Workloads

In Fig. 5.15, the ProteusTM system is evaluated as a whole. Both RecTM and PolyTM amount to 6.5K and 6K lines of code, excluding third party code (e.g., Mahout and TMs).

4 use cases for ProteusTM’s runtime optimization are shown on 2 TM benchmarks (Red-Black Tree and STMB7 [52]), a TM porting of TPC-C [88] and of Memcached [78]. For each application 3 workloads are triggered and chosen to exemplify contrasting characteristics and resulting performances. In each case, ProteusTM is totally oblivious of the target application: no workloads of the application are present in its training set. This highlights the Recommender’s ability to detect similarity patterns between the target workloads and the set of disjoint applications used as training set.

Setting the Monitor period to 1 sec and the SMBO \( \epsilon \) to 0.01. In each run, let us measure the performance of (i) ProteusTM, (ii) the 3 configurations that perform best in each workload, (iii) the Best Fixed configuration on Average (BFA) across the workloads, and (iv) a Sequential non-instrumented execution.

Three conclusions can be drawn from these plots: (i) ProteusTM is able to quickly identify, at runtime, configurations that are optimal — or very close. Remarkably, ProteusTM delivers performance that is, on average, only 1% lower than the optimal; (ii) employing any of the baseline alternatives yields up to two orders of magnitude lower performance; (iii) thanks to the SMBO approach, the performance degradation incurred when exploring is minimal (at most 7 explorations in these use cases). Such cost is usually amortized in long-running services (e.g., databases), in which workload shifts are infrequent [22].

A summary is provided in Table 5.10 where it is listed the optimal configurations in each workload. It also shows the BFA (with ⭐) which is always also an optimal configuration in some workload.

This data highlights the robustness of ProteusTM to optimize heterogeneous applications with diverse optimal configurations, in terms of TM algorithm (STMB7), parallelism degree (TPC-C) and HTM tuning (RBT and Memcached).

Finally, in Fig. 5.16, what was claimed in Section 4.3.3 is confirmed by using a static TPC-C workload and varying external factors to the application to trigger behavior changes. To simulate these external changes the stress Unix tool was used with different configurations over periods of 30 seconds: it either created high CPU, memory or IO usage in each workload. The results are similar to what was chosen previously, in that ProteusTM obtains perform close to the optimal across the test.
## Table 5.10: For each benchmark (of Fig. 5.15), it is showed the MDFO (in %) of ProteusTM, each Optimal and BFA (★) configurations. Each workload is labeled with its optimal configuration.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Benchmark</th>
<th>Workload (Opt Conf)</th>
<th>Optimal in Workload</th>
<th>ProteusTM (explorations)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Opt 1</td>
<td>Opt 2</td>
</tr>
<tr>
<td>A</td>
<td>RBT</td>
<td>1 (NOrec: 7t)</td>
<td>0</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 ★ (HTM: 8t Half-20)</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 (HTM: 4t GiveUp-4)</td>
<td>154</td>
<td>37</td>
</tr>
<tr>
<td>A</td>
<td>STMB7</td>
<td>1 (HTM: 4t Linear-2)</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 (Swiss: 4t)</td>
<td>135</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 ★ (TL2: 8t)</td>
<td>390</td>
<td>29</td>
</tr>
<tr>
<td>A</td>
<td>TPC-C</td>
<td>1 ★ (Tiny: 4t)</td>
<td>0</td>
<td>273</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 (HTM: 3t GiveUp-16)</td>
<td>68</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 (Tiny: 8t)</td>
<td>22</td>
<td>370</td>
</tr>
<tr>
<td>B</td>
<td>Memchd</td>
<td>1 ★ (Swiss: 32t)</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 (Tiny: 32t)</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 (Tiny: 4t)</td>
<td>18</td>
<td>66</td>
</tr>
</tbody>
</table>

**Figure 5.16:** Similar to Fig. 5.15c, but with a static application workload, varying instead the availability of machine resources.
Chapter 6

Conclusion

Transactional Memory is a prominent alternative to classic, lock-based synchronization schemes. The TM abstraction provides programmers a simple way for developing concurrent applications. Using TM, programmers only need to specify what code blocks should be executed atomically, leaving to the TM library the burden of implementing how to achieve it.

Over the last years, many TMs have been proposed in literature. More recently, the relevance of TM was amplified by the integration of hardware support in processors by Intel and IBM.

Although TM simplifies the development and verification of concurrent programs, the performance efficiency of state of the art TMs highly depends on the workload characteristics, and the employed hardware.

This has raised interest in designing self-tuning solutions that adapt specific building blocks of existing TM implementations.

These solutions, however, only support the adaptation of individual aspects of a TM implementation, such as selecting the right TM algorithm, tuning some of its internal parameters, or adapt the degree of parallelism.

In this dissertation it is presented ProteusTM, a multi-dimensional self-tuning TM that aims to explore global optimal configurations through the adaptation of multiple individual TM parameters. ProteusTM encompasses two main modules: PolyTM and RecTM. PolyTM is a software library containing a range of alternative TM implementations (including 4 STMs as well as Intel HTM) and mechanisms for dynamically switching among them. RecTM is the module that acts as a recommender, which receives monitored KPI, and give its prediction regarding the best configuration for the current workload.
An extensive sensitivity analysis study was conducted on the TM parameters. This study not only served to motivate the need for self-tuning, but also allowed to identify the most important configuration parameters of TM implementations.

ProteusTM has been evaluated via an extensive study using 15 TM applications, a parameter space of up to 130 configurations, and optimizing 2 metrics: performance and energy efficiency. The results highlight that ProteusTM obtains quasi-optimal performance (on average $< 3\%$ from optimal) and gains up to 2 orders of magnitude over static alternatives.

ProteusTM does not solve all challenges of TM (e.g., coping with external side-effects in transactions), but it drastically improves TM performance while preserving its simplicity.

### 6.1 Future Work

The work started with this dissertation can be extended in several directions:

- Currently, ProteusTM can perform adaptations along three dimensions: parallelism degree, TM backend, HTM retry parameters. Another possibility would be to dynamically adjust the mapping of threads to (physical/virtual) cores, especially in NUMA architectures.

- The results of the sensitivity analysis presented in 3 highlight that the relevance of contention management becomes marginal if one can dynamically adjust the degree of parallelism (and the TM implementation) to fit the workload characteristics. It would be interesting to verify whether this experimental finding is confirmed even when considering architectures capable of supporting higher levels of parallelisms than the ones available for my experimental study at the moment of writing this dissertation.

- In the study, a possible search space consisting of a quite large number of alternative configurations was considered. It would be interesting to analyze the gains achievable by restricting the potential number of configurations available to a smaller subset. The idea here is to narrow the search space to include solely high quality configurations, i.e., configurations that have a high chance to be quasi-optimal (e.g., $5\%$ from optimum).

- Finally, a future study should investigate how the sparsity of the training data can impact the prediction accuracy. More in detail, it would be interesting to verify the sensitivity of the proposed collaborative filtering techniques to scenarios in which, for the workloads in the training set, only a subset of configurations is actually known.
Bibliography


[70] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanawamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for


