UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

UNDER ERASMUS MUNDUS PROGRAMME

TECHNIQUES FOR ENHANCING THE EFFICIENCY OF TRANSACTIONAL MEMORY SYSTEMS

SHADY ALAAELDIN MOHAMED ABDELKADER RABIE ISSA

Supervisor: Doctor PAOLO ROMANO
Co-Supervisor: Doctor VLADIMIR VLASSOV

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

Jury final classification: Pass with Distinction

2018

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

UNDER ERASMUS MUNDUS PROGRAMME

TECHNIQUES FOR ENHANCING THE EFFICIENCY OF TRANSACTIONAL MEMORY SYSTEMS

SHADY ALAAELDIN MOHAMED ABDELKADER RABIE ISSA

Supervisor: Doctor PAOLO ROMANO
Co-Supervisor: Doctor VLADIMIR VLASSOV

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

Jury final classification: Pass with Distinction

Jury

Chairperson:
Doctor João Emílio Segurado Pavão Martins, Instituto Superior Técnico, Universidade de Lisboa

Members of the Committee:
Doctor Konstantin Busch, School of Electrical Engineering and Computer Science, Louisiana State University, USA
Doctor António Manuel Ferreira Rito da Silva, Instituto Superior Técnico, Universidade de Lisboa
Doctor Paolo Romano, Instituto Superior Técnico, Universidade de Lisboa
Doctor Tatiana Shpeisman, Google Brain, USA

Funding Institutions
European Commission
Fundação para a Ciência e Tecnologia

2018

Resumo

Memória Transacional (TM) é um paradigma emergente de programação concorrente que permite simplificar enormemente o desenvolvimento de aplicações concorrentes ao aliviar os programadores de uma grande e complexa tarefa: como assegurar uma correta e eficiente sincronização de acessos a memória partilhada. Apesar do grande número de estudos recentes dedicados a esta área, os sistemas TM ainda sofrem de graves grandes limitações que dificultam limitam tanto a sua performance como a sua eficiência energética.

Esta tese aborda o problema de como construir implementações eficientes desta abstração, nomeadamente a TM, ao introduzir técnicas inovadoras que discutem três limitações cruciais dos sistemas TM existentes: *(i)* estender a capacidade efetiva das implementações de TM em hardware (HTM); *(ii)* reduzir os custos de sincronização nos sistemas de TM híbridos (HyTM); *(iii)* melhorar a eficiência das aplicações de TM via esquemas de gestão de contenção energeticamente cientes.

POWER8-TM (P8TM) é a primeira contribuição desta tese e aborda uma das mais convincentes maiores limitações das implementações HTM: a inabilidade impossibilidade de processar transações cuja quantidade de dados lidos e/ou escritos exceda a capacidade da memória cache do processador. P8TM aproveita de uma forma inovadora duas características de hardware providenciadas pelos processadores POWER8 da IBM, nomeadamente as Transações Rollback-only e o mecanismo de Suspensão/Retoma. P8TM pode alcançar até 7× mais ganhos na sua performance em cenários que testam os limites da capacidade da HTM.

A segunda contribuição é Dynamic Memory Partitioning-TM (DMP-TM), uma nova

implementação HyTM que delega o custo de deteção de conflitos entre HTM e TM baseada em software (STM) a mecanismos de proteção de memória do hardware e do sistema operativo. O design do DMP-TM é independente do algoritmo usado para regular transações STM e tem a grande vantagem básica de permitir a integração, de uma forma eficiente, de quaisquer implementações STM que permitam uma grande escalabilidade que, de outra forma, exigiriam uma instrumentação dispendiosa sobre as transações HTM. Isto permite ao DMP-TM alcançar ganhos até 20× comparativamente a outras soluções do estado da arte da HyTM em cenários com pouca contenção nos acessos.

Green-CM, um gestor de conflitos (CM) com consciência energética, é a terceira contribuição desta tese e apresenta dois aspetos inovadores: *(i)* um novo design assimétrico que combina diferentes políticas de *back-off*, retirando vantagem das capacidades de hardware da Frequência Dinâmica e do Escalonamento de Voltagem (DVFS) disponível na maioria dos processadores modernos; *(ii)* uma implementação com eficiência energética de um componente fundamental para muitas implementações de CM, nomeadamente, o mecanismo usado para fazer *back-off* de tarefas por um tempo predefinido. Graças ao seu inovador design, Green-CM pode reduzir o Produto do Atraso Energético até 2.35× relativamente aos CMs mais inovadores.

Todas as técnicas propostas nesta dissertação partilham uma importante característica comum essencial para preservar a facilidade de uso da abstração que é a TM: a utilização, em tempo de execução, de mecanismos de auto- configuração que assegurem uma performance robusta mesmo em presença de uns cenários heterogéneos, sem requerer qualquer conhecimento prévio dos cenários alvo ou da sua arquitetura.

Abstract

Transactional Memory (TM) is an emerging programming paradigm that drastically simplifies the development of concurrent applications by relieving programmers from a major source of complexity: how to ensure correct, yet efficient, synchronization of concurrent accesses to shared memory. Despite the large body of research devoted to this area, existing TM systems still suffer from severe limitations that hamper both their performance and energy efficiency.

This dissertation tackles the problem of how to build efficient implementations of the TM abstraction by introducing innovative techniques that address three crucial limitations of existing TM systems by: *(i)* extending the effective capacity of Hardware TM (HTM) implementations; *(ii)* reducing the synchronization overheads in Hybrid TM (HyTM) systems; *(iii)* enhancing the efficiency of TM applications via energy-aware contention management schemes.

The first contribution of this dissertation, named POWER8-TM (P8TM), addresses what is arguably one of the most compelling limitations of existing HTM implementations: the inability to process transactions whose footprint exceeds the capacity of the processor's cache. By leveraging, in an innovative way, two hardware features provided by IBM POWER8 processors, namely Rollback-only Transactions and Suspend/Resume, P8TM can achieve up to 7× performance gains in workloads that stress the capacity limitations of HTM.

The second contribution is Dynamic Memory Partitioning-TM (DMP-TM), a novel Hybrid TM (HyTM) that offloads the cost of detecting conflicts between HTM and Software

TM (STM) to off-the-shelf operating system memory protection mechanisms. DMP-TM's design is agnostic to the STM algorithm and has the key advantage of allowing for integrating, in an efficient way, highly scalable STM implementations that would, otherwise, demand expensive instrumentation of the HTM path. This allows DMP-TM to achieve up to 20× speedups compared to state of the art HyTM solutions in uncontended workloads.

The third contribution, Green-CM, is an energy-aware Contention Manager (CM) that has two main innovative aspects: *(i)* a novel asymmetric design, which combines different back-off policies in order to take advantage of Dynamic Frequency and Voltage Scaling (DVFS) hardware capabilities, available in most modern processors; *(ii)* an energy efficient implementation of a fundamental building block for many CM implementations, namely, the mechanism used to back-off threads for a predefined amount of time. Thanks to its innovative design, Green-CM can reduce the Energy Delay Product by up to 2.35× with respect to state of the art CMs.

All the techniques proposed in this dissertation share an important common feature that is essential to preserve the ease of use of the TM abstraction: the reliance on on-line self-tuning mechanisms that ensure robust performance even in presence of heterogeneous workloads, without requiring prior knowledge of the target workloads or architecture.

Palavras-chave

Memória Transacional, Programação Paralela, Eficiência Energética, Mecanismos de Auto-configuração, Partição de Dados, Frequência Dinâmica e do Escalonamento de Voltagem (DVFS), Memória Transacional em Hardware (HTM)

Keywords

Transactional Memory, Parallel Programming, Concurrency Control, Self-tuning, Energy Efficiency, Data Partitioning, Dynamic Frequency and Voltage Scaling (DVFS), Hardware Transactional Memory (HTM)

Acknowledgments

One of the hassles that I have been worrying about, since the moment I felt I would reach this point, is how to express my gratitude towards Prof. Paolo Romano. Looking back at these years, I seriously doubt I could have made it so far if Paolo was not my supervisor. I remember at one point down the road, where I would have gone out of track if it was not to his patience and support. Although he insists it is his duty, he takes it one mile further, not just one step. He always impresses me by how relentless and persistent he is (touch wood), how much he aspires and pushes to do work as perfect as possible — I believe this is the true essence of being a researcher. I have been growing both personally and professionally with each discussion we have. I feel very lucky that I got to be your student. I will always owe many things to you, Grazie Mille Paolo!

I would like to thank Prof. Vladimir Vlassov who thankfully accepted to serve as my supervisor at KTH at a very critical time. I am also very grateful for the opportunity to work with Prof. Pascal Felber and Alexander Matveev, working with such great minds is an honor and a joy. Although we never met physically, in the virtual world, they broadened my perspectives, directly for Switzerland and United States, motivating me to become a better researcher.

From INESC-ID I want to thank Nuno Diegues whose help at the beginning of my PhD made it much easier along the way. I also want to thank Amin Mohtasham, who made my landing in Portugal very smooth. All the chats, food (Rodizios to be more specific), discussions, going out and football made my stay in Portugal more enjoyable. I would also like to extend my thanks to Amin Khan, Manuel Bravo, Pedro Raminhas and Daniel Castro.

This journey would not have been bearable without my Egyptian friends across Europe, Bakr, Hatem, Karim, Loay, Mahmoud, Obda and Walid. After finishing our Master at Nile University, we all started similar journeys, but in different countries. The all-day-long availability, our visits to each other and the trips we went for added another dimension making our experience richer and more meaningful. I would like to especially thank Walid and Mahmoud, with whom the journey started much earlier during the Bachelor at Cairo University, for the strength their companionship provides. I also want to send special thanks to Karim, the friend who is always there, who helps even before it is needed.

The same goes for my siblings and cousins, Aboody, Shadwa, Mohamed, Ahmed, Marwan, Dodi and Khaled whom despite the distance, never made me feel far away. Our continuous connection helped not to feel homesick. I also would like to thank Hasan for

the genuine friend he is. I really appreciate how he gets out of his way during my short visits to Egypt to catch up, it always made me feel as I have never left.

Now it comes to the person who was pushing and supporting from the other side of Paolo, Patrícia, without whom I would not be writing this now. She is the one who deals with all my nincompoops on a daily basis, and for that I am very grateful for her patience and understanding. Above all, she gives me structure and shines light through my life. I feel blessed to have you in my life, Batata.

Last, but not the least, I think I will never be able to rightfully thank my parents, Alaa and Nahla, who sacrificed literally everything for the sake of my well-being and my siblings'. Being the eldest son, I have seen how you spared no effort to make our lives as smooth as you can while preparing us for the tough journey of life. I will be eternally grateful for all what you have done.

The work done in this thesis was funded by European Commission (EACEA) through the Erasmus Mundus doctoral fellowship, via Erasmus Mundus Joint Doctorate in Distributed Computing (EMJDDC) programme. It was also supported by Portuguese national funds through Fundação para a Ciência e Tecnologia (FCT), via the projects UID/CEC/50021/2013, EXPL/EEI-ESS/0361/2013 and PTDC/EEISCR/1743/2014.

Contents

Contents xvii

List of Figures xxi

List of Algorithms xxiii

1 Introduction 1
- 1.1 Problem Definition . . . 2
- 1.2 Research Questions . . . 3
- 1.3 Research Methodology . . . 4
  - 1.3.1 Self-tuning . . . 4
  - 1.3.2 Evaluation . . . 4
  - 1.3.3 Challenges . . . 5
  - 1.3.4 Limitations . . . 6
- 1.4 Thesis Contributions . . . 6
- 1.5 List of Publications . . . 8
- 1.6 Structure of this Document . . . 9

2 Background 11
- 2.1 Conventional Synchronization Mechanisms . . . 11
  - 2.1.1 Mutual Exclusion . . . 11
  - 2.1.2 Read-Copy-Update . . . 13
- 2.2 Transactional Memory . . . 14
  - 2.2.1 Abstraction . . . 14
  - 2.2.2 Guarantees . . . 15
  - 2.2.3 Design Choices . . . 16
  - 2.2.4 Software Transactional Memory . . . 16
  - 2.2.5 Hardware Transactional Memory . . . 18
  - 2.2.6 Hybrid Transactional Memory . . . 20
  - 2.2.7 Contention Management and Scheduling . . . 22
- 2.3 Self-tuning for Transactional Memory . . . 24
- 2.4 Benchmarks for Transactional Memory . . . 25

3 POWER8-TM **27**
3.1 Problem . . . 27
3.2 Overview . . . 28
3.3 Description . . . 29
3.3.1 Uninstrumented Read-Only Transactions . . . 29
3.3.2 Touch-based Validation . . . 31
3.4 Algorithm . . . 32
3.4.1 URO Path . . . 32
3.4.2 ROTs Path . . . 34
3.4.3 Complete Algorithm . . . 34
3.4.4 Correctness Argument. . . . 38
3.5 Read-set Tracking . . . 39
3.6 Self-tuning . . . 40
3.7 Evaluation . . . 42
3.7.1 Read-set Tracking . . . 42
3.7.2 Sensitivity Analysis . . . 44
3.7.3 STAMP Benchmark Suite . . . 48
3.7.4 TPC-C Benchmark . . . 52
3.8 Related Work . . . 52
3.9 Summary . . . 54

4 DMP-TM **55**
4.1 Problem . . . 55
4.2 Overview . . . 56
4.3 Description . . . 57
4.3.1 Memory Manager: Double Heap Approach . . . 58
4.3.2 Memory Manager: Enforcing Dynamic Partitions . . . 59
4.3.3 Transaction Scheduler and Auto-Tuner . . . 61
4.4 Algorithm . . . 61
4.4.1 Correctness Argument . . . 64
4.4.2 Optimizations . . . 65
4.5 Evaluation . . . 66
4.5.1 Synthetic Benchmarks . . . 66
4.5.2 STAMP Benchmark Suite . . . 69
4.5.3 TPC-C Benchmark . . . 72
4.6 Related Work . . . 74
4.7 Summary . . . 75

5 Green-CM **77**
5.1 Problem . . . 77
5.2 Overview . . . 77
5.3 Description . . . 79
5.3.1 Hybrid Back-off Implementation . . . 79

5.3.2 Asymmetric Contention Management 81
5.3.3 Controller 83
5.4 Evaluation 86
5.4.1 Tuning Strategies 87
5.5 Related work 92
5.6 Summary 94

6 Final Remarks **95**
6.1 Future work 97

Bibliography **99**

List of Figures

3.1 In order to preserve consistency, the write back of shared variables updated by an update transaction must be delayed until after any URO transaction has completed execution. 30
3.2 A read access to a shared variable updated by a suspended update transaction will abort the latter (when it resumes). 30
3.3 ROTs do not track reads and may, as such, observe different values when reading the same variable multiple times. 31
3.4 By re-reading x during *rot-rset* validation at commit time (denoted by v:r), T_1 forces the abort of T_2 that has updated x in the meantime. 31
3.5 Different execution paths that can be used by transactions and rules for switching between them. Lines within rules represent necessary memory barriers. . . 41
3.6 Evaluation of different implementations of read-set tracking. 43
3.7 High capacity-low contention configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios. 45
3.8 High capacity, high contention configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios. 47
3.9 Low capacity, low contention configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios. 48
3.10 Throughput, abort rate, and breakdown of commit modes of STAMP benchmarks (1). 49
3.11 Throughput, abort rate, and breakdown of commit modes of STAMP benchmarks (2). 50
3.12 Throughput, abort rate, and breakdown of commit modes of TPCC at 10%, 50% and 90% update ratios. 53

4.1 Architecture of DMP-TM. 57
4.2 Mapping of the address space in the HTM Heap and in the STM Heap. 59
4.3 Speedup and commits breakdown for disjoint data structures running 1, 10 and 80 threads. 67
4.4 Speedup, commits breakdown and system calls ratio for non-disjoint data structure running 1, 10 and 80 threads. 70

4.5 Speedup, commits breakdown and system calls ratio for *genome* and *intruder* of STAMP benchmark suite. 72
4.6 Speedup, commits breakdown and system calls ratio for two workloads of TPC-C. 73

5.1 Architecture of Green-CM. 80
5.2 EDP of different static configurations of α with $\mathcal{T} = min\_sleep$, normalized with respect to the EDP of the best, off-line identified, configuration for α and $\mathcal{T}$. 84
5.3 EDP for different static values of $\mathcal{B}$ normalized to the EDP obtained using the best setting of $\mathcal{B}$. 84
5.4 Normalized EDP across different benchmarks using different strategies for self-tuning individually α. 87
5.5 Normalized EDP across different benchmarks using different strategies for self-tuning individually $\mathcal{B}$. 88
5.6 Normalized EDP across different benchmarks using different strategies for coupling the two tuners. 89
5.7 EDP, energy consumption and execution time of *intruder* (left) and *kmeans* (right), normalized with respect to Green-CM (Higher is better). 90
5.8 EDP, energy consumption and commit rate of STMbench7 (left) and Memcached (right), normalized with respect to Green-CM (Higher is better). 91
5.9 EDP, energy consumption and time for *intruder* with and without ACM enabled. 92
5.10 Frequency distribution of different thread configurations for *intruder*. 93

List of Algorithms

1 P8TM: URO path only algorithm 33
2 P8TM: ROT path only algorithm 35
3 P8TM: complete algorithm 36
4 P8TM: complete algorithm (2) 37
5 P8TM: URO path 38
6 DMP-TM: pseudocode for STM 62
7 DMP-TM: pseudocode for signal handler 63
8 Green-CM: hybrid back-off mechanism 81

Chapter 1

Introduction

For decades, processor manufacturers have been able to double the computational power of CPUs approximately every 18 months — a fact that goes commonly under the name of Moore's Law [1, 2]. Initially, such an exponential growth has been supported by a continuous increase in the clock frequency of processors. However, at some point processor manufacturers were unable to further scale up the clock frequency of processors due to thermal issues. This forced a paradigm shift towards having several cores that can operate in parallel inside the same processor, marking the beginning of the multicore era. Nowadays, multicore processors represent the dominant architecture across all devices ranging from servers to mobile devices, and the trend is towards having tens to hundreds of cores integrated in the same platform.

In order to cope with this shift in processors' design, software development has also accordingly undergone a transition from serial to parallel programming — a notorious harder problem. One of the key difficulties associated with parallel programming is how to handle data races, which occur when threads running on different cores access the same data at the same time, and at least one of them updates it. Without proper inter thread synchronization, in these cases, the behavior of the program becomes unpredictable.

The traditional way of enforcing synchronization is by using mutexes or locks to protect shared data. However, the inherently sequential nature of locks limits concurrency and can severely hinder performance in large scale parallel systems. One way to minimize the drastic effects of locking is to use fine-grained locking. Programs using fine-grained locking schemes regulate logically independent accesses to different shared memory regions via multiple locks. By allowing threads to acquire only locks protecting data that they are going to access, fine-grained locking can enable high degrees of parallelism. Unfortunately, though, they are also notorious for being complex to devise, debug and reason about. They are also infamous for suffering of subtle pathologies (e.g., deadlocks, livelocks and priority inversions) that are complex to reason about and reproduce as well as for compromising a property that is crucial for modern software systems, namely composability [3].

Transactional Memory (TM) is an attractive, alternative synchronization paradigm for parallel programming that preserves the simplicity of coarse-grained locking, while striv-

ing to achieve performance levels on par, or even superior, to the ones attainable via the use of complex fine-grained locking schemes. The main driver of TM lies in its simplicity: TM has borrowed the semantics of atomic transactions from the database literature and applied them to the concurrent programming domain, providing programmers with a familiar abstraction to synchronize concurrent accesses to shared memory regions. With TM, programmers need to simply devise their code into atomic blocks, or transactions, delegating to the underlying TM system the problem of guaranteeing their atomicity. Recently, TM has arguably made significant progresses towards mainstream adoption, thanks to the achievement of an important milestone: the official integration of programming language supports in two mainstream languages, namely C and C++ [4].

1.1 Problem Definition

Herlihy and Moss [5] initially proposed TM as a hardware modification to the cache coherence protocol for multi-processor systems. However, only since the early 2000s, with the advent of multi-core processors in mainstream architectures, that research in the area of TM started to gain momentum. Since then, TM was the object of intense research efforts. Unlike in its original formulation, though, most of the research during the first decade of 2000 was focused on Software Transactional Memory (STM) — mainly due to the lack of availability of commercial hardware platforms supporting the TM abstraction and to the portable and flexible nature of software. This resulted in a plethora of algorithms [6–10], which aim to optimize STM performance when faced with different workloads, but that still incur non-negligible instrumentation overheads due to the need for tracking transactional accesses via additional software mechanisms [11–13].

Only recently, in 2012 specifically, IBM provided the first commercially available implementation of Hardware Transactional Memory (HTM) in several processor families: POWER [14], Blue Gene P/Q [15] and system Z [16]. This was not the first attempt though: just a few years earlier, Sun declared a processor called Rock that has support for HTM [17], but it never reached the market. AMD also released specifications and instructions set extension for HTM [18] that were never implemented by any of its processors. More recently, Intel, starting from the Haswell family, has integrated in many of its CPUs supports for HTM, which was named TSX [19].

Similarly to the initial proposal, existing HTM systems are implemented as an extension to the cache coherence protocol. This design choice has simplified the problem of integrating HTM in existing processors. However, though, it has also intrinsically exposed HTM to crucial limitations. One of the most crucial limitations is that, since with HTM read-sets and write-sets are tracked in processor's cache, transactions whose read-set and write-set do not fit in the cache can never commit in hardware, even in absence of data conflicts. Other limitations include their inability to ensure atomicity of transactions that issue prohibited instructions (e.g., system calls), or that incur page-faults and timer interrupts [20, 21]. Thus, in order to ensure progress, HTM must be complemented with a fallback synchronization mechanism, implemented in software. The typical fallback mechanism for

HTM consists in acquiring a single global lock - an approach that has the advantage of being simple and generic, but that imposes a severe performance toll as the activation of the fallback aborts any concurrent transaction and prevents any parallelism.

Hybrid Transactional Memory (HyTM) [22, 23] were conceived to provide HTM with a more scalable fallback path, by enabling the concurrent use of HTM and STM. Conceptually, this can bring the merits of both worlds: efficiency of HTM and robustness of STM. Unfortunately, though, existing HyTM implementations [24–28] incur expensive instrumentation overheads in order to synchronize concurrent transactions executing in hardware and software [29].

Another known issue of TM, which can be seen as oblivious to its implementation and to its optimistic nature, is that workloads with high degree of contention generate frequent transaction aborts. This can have a detrimental impact not only on the performance of TM applications, but also on their energy efficiency — a factor that is increasingly relevant for a wide range of systems, from sensors and mobile devices to data centers [30]. Quite surprisingly though, despite the abundant research on TM, the problem of how to design energy efficient TM has been largely overlooked in the literature.

1.2 Research Questions

TM systems have managed to fulfill their first premise of being easy to use [31]. However, they still fall behind on delivering satisfactory efficiency levels, due to several reasons such as: the instrumentation overheads of STM, the intrinsic limitations of HTM, the high synchronization costs incurred by HyTM systems and the overlooking of energy efficiency in their designs as explained in the previous section. Accordingly, the problem of enhancing the efficiency of TM systems, without compromising their ease of use, remains a relevant and open research question. This dissertation aims to bridge the gap towards solving this problem, by answering the following questions:

1. How to extend the effective capacity of existing HTM systems?

   Unlike STM, HTM does not suffer from instrumentation overheads. However, the performance of HTM systems can be hampered by their inability to execute transactions that do not fit within their hardware limits. Due to the high cost of hardware modifications, this limitation is not expected to be solved in the future. Therefore, to increase the usefulness of HTM systems, it is crucial to develop novel mechanisms that enable existing HTM implementations to accommodate larger transactions.

2. How to reduce the synchronization costs incurred by HyTM systems?

   Even with novel mechanisms mitigating the intrinsic limitations of HTM, a software-based fallback synchronization mechanism is always needed to guarantee forward progress. HyTM systems that try to complement the best-effort HTM systems with robust STM implementations pay high synchronization costs to ensure correct concurrent execution of HTM and STM transactions. Accordingly, reducing these costs is crucial for enhancing the efficiency of TM systems.

3. How to design energy-aware contention management for TM systems?

   Contention Manager (CM) modules, which are typically integrated within TM systems, aim to reduce the detrimental effects of contention. With the increasing importance of energy efficiency and aborted transactions being a main source of wasted energy, a key open question is how to design CM modules aiming not only to maximize performance, but also incorporate energy efficiency as a first-order optimization goal.

1.3 Research Methodology

To tackle the problems described above, the work of this dissertation followed an empirical approach, which entails iterations of the following steps: *(i)* quantifying an identified problem; *(ii)* proposing a solution; *(iii)* ensuring its correctness; *(iv)* evaluating the proposed solution; and *(v)* analyzing the results. Upon iterating these steps, the problems, when feasible, were dissected into several sub-problems, each going through its own iterations, before merging into a single solution in the end. The following few subsections overview the approach followed in this dissertation regarding incorporating self-tuning mechanisms, evaluation of contributions, the challenges that were met throughout the coursework and the scope of the thesis in general.

1.3.1 Self-tuning

One aspect that is common across all the contributions of this dissertation is the extensive use of self-tuning techniques. The use for self-tuning in TM is not new and stems from the high dependability of the design choices on workload and architecture characteristics. To avoid affecting the ease of use of TM, which is a cornerstone for the thesis of this dissertation, the incorporated self-tuning mechanisms had to satisfy two critical features: online and transparent . This is necessary to avoid the need of any change to the abstraction provided by TM to the programmers. Furthermore, these mechanisms must be lightweight in order not to outweigh the performance gains in favorable settings and to pay only a small penalty in non-favorable ones.

A brute force approach was followed to configure the chosen self-tuning mechanisms. The different design choices were identified and the search space of each was explored systematically. Then, to combine these design choices, various policies were considered according to the dependency relationships among the different parameters. Finally, the best on average configuration was chosen.

1.3.2 Evaluation

The proposed contributions were evaluated against state of the art TM systems using publicly available implementations. TM implementations were integrated within the same framework with a collection of numerous benchmarks. All TM systems were linked with the benchmarks using the same interface and were compiled on the same platform using

the same compiler flags and external library for memory allocation [32]. All these implementations are open sourced in public repositories, alongside the configurations used in the experiments [33–35], in order to ease the reproducibility of the results included in this dissertation.

All experiments involving hardware features, such as HTM, were executed using existing commercial systems, without relying on any simulation or emulation environments. The experiments were executed using automated scripts to ensure their reproducibility. The reported results are the average of at least 5 runs with a standard deviation less than 10% of the mean value. To compare performances, throughput and execution times were measured, while for energy, data from processor registers to track energy consumption were collected. Other than that, commits and aborts breakdowns were collected to provide a more in-depth understanding of the results.

Experiments were conducted using microbenchmarks and real-life applications. Microbenchmarks consisted of concurrent datastructures that support operations such as lookups, inserts and deletes. This allows for devising sensitivity studies that put focus on different aspects of a design: impact of different components of the solution, the potential gains in ideal scenarios and overheads in the worst cases. Real-life applications that span a breadth of diverse application domains ranging from multi-threaded servers to OLTP systems and CAD like applications assessed the performance of the presented solutions in complex settings. Section 2.4 describes the benchmarks used throughout the thesis in more detail.

1.3.3 Challenges

All the contributions of this dissertation share the common feature of exploiting hardware features available in recent processors. This requires developing hardware-aware software, which comes with several challenges. Controlling the hardware explicitly from the userspace may be very expensive due to the need for issuing system calls. The high cost of systems calls may outweigh gains achievable by tweaking the hardware configuration, which required incorporating mechanisms to optimize such trade off. The approach followed in this dissertation to avoid paying the toll of system calls is by exploiting automatic hardware re-configuration. However, this comes with other challenges, mainly the fact that hardware manufactures do not disclose all the details of how these automatic hardware mechanisms operate. To overcome this obstacle, various experimental studies were conducted to understand the underlying mechanisms.

Another major challenges, which is common for parallel programming in general, and HTM in specific is debugging due to the non-deterministic nature of concurrent software. With HTM, it is not even possible to halt the execution within a transaction and investigate the state of the system. Moreover, upon an abort, all changes within a transaction are discarded and the system is rolled back to state before the transaction started, leaving only some indication of the cause of the abort. As a consequence, developing HTM programs, with current limited debugging support, entails a very tedious process of inspecting each piece of code in an isolation of each other.

1.3.4 Limitations

The scope of this work is limited to parallel software that can benefit from TM as a synchronization mechanism. This can be either programs built directly using the TM abstraction, or ones that use classical synchronization mechanisms implemented using TM (e.g. lock elision, see Section 2.2.5). In general, non-TM friendly parallel software, for example, software with frequent activation of critical sections that execute irrevocable operations (e.g. I/O), may not benefit from this work.

Further, each contribution presented in this thesis has its own limitations. These limitations are detailed in the chapters describing these solutions and can range from workloads where they are not beneficial to architectures that they do not support. Nevertheless, they are augmented with self-tuning mechanisms to fall back to a baseline that perform on par with the state of the art. Chapter 6 discusses ideas for future work that can be pursued to achieve gains in several of these scenarios.

1.4 Thesis Contributions

These three research questions discussed in Section 1.2 are addressed via three novel contributions, which are overviewed next.

POWER8-TM (P8TM) (Chapter 3) is a novel TM that aims to expand the actual capacity available in HTM systems by exploiting two micro-architectural features available in the IBM POWER8 HTM implementation: Suspend/Resume (S/R) and Rollback-Only Transactions (ROTs). As the name suggests, the earlier allows for suspending a transaction, executing non-transactional code, and then resuming the transactions execution. ROTs are a special type of transactions supported by POWER8 HTM that are meant to provide failure-atomicity, i.e., the ability to roll-back the execution of a code block, but they do not track read memory accesses in hardware. As a consequence, ROTs have a larger capacity than normal transactions, but they do not guarantee safety in presence of concurrent executions.

P8TM relies on several complementary mechanisms. The first one consists in executing read-only transactions without instrumentation and outside the context of hardware transactions. Such design, spares read-only transactions from any capacity limitation, as well as other HTM restrictions (e.g., spurious aborts due to context switches), hence, bringing significant gains for read-dominated applications. To preserve correctness of read-only transactions, P8TM relies on a scheme similar in spirit to the Read Copy Update (RCU) quiescence mechanism [36–38], which ensures that update transactions, executing in hardware, can only commit if there are no concurrent active read-only transactions. Second, to accommodate large update transactions in hardware, P8TM incorporates a novel algorithm called Touch-To-Validate (T2V). T2V enables the safe execution of concurrent ROTs, which allows P8TM to exploit ROT's ability of fitting larger transactions to mitigate the capacity constraints of update transactions running in hardware.

To ensure robust performance in workloads that do not exhibit capacity aborts, i.e.,

where plain HTM would excel, P8TM employs a light-weight, yet effective, self-tuning mechanism to determine when to disable the quiescence and T2V mechanisms in order to avoid unnecessary overheads. This allows P8TM to achieve up to ∼7× throughput gains with respect to plain HTM and extend capacity of update transactions by more than one order of magnitude, while remaining competitive even in unfavorable workloads.

Dynamic Memory Partitioning-TM (DMP-TM) (Chapter 4) is a novel HyTM algorithm that exploits the idea of leveraging operating system-level memory protection mechanisms to detect conflicts between HTM and STM transactions. Thanks to this design approach, DMP-TM is capable of supporting highly scalable STM implementations, whose integration in a HyTM system would otherwise require expensive instrumentations of the HTM path, without any instrumentation on the HTM path. The key challenge related to DMP-TM's design is that it relies on system calls to enforce memory partitions, which have a non-negligible cost. Indeed, DMP-TM investigates an interesting trade-off: leveraging the data partitionability present in applications in order to reduce the run-time overheads of detecting conflicts among STM and HTM transactions, at the cost of a performance penalty in case conflicts between STM and HTM transactions do materialize.

In order to maximize the gains achievable in favorable workloads, while ensuring robust performance also with unfavorable ones, DMP-TM integrates two key self-tuning mechanisms that detect, in a transparent and automatic way: *(i)* which back-end (STM or HTM) to employ for the different transactional blocks of a TM application; *(ii)* whether the degree of partitionability of the accesses generated by the STM and HTM back-ends is too low, being thus preferable to use exclusively the most efficient of the two back-ends. Compared with state of the HyTM systems, DMP-TM demonstrated robust performance in an extensive evaluation achieving gains of up to ∼20×.

Green-CM (Chapter 5) is a CM scheme explicitly designed to jointly optimize both performance and energy consumption. It pursues this goal via two novel complementary mechanisms.

The first innovative mechanism of Green-CM consists in an energy efficient implementation of a key building block of many contention management schemes, namely the `wait` primitive used to back off threads, upon a conflict, for some predetermined period of time. The wait primitive can, in practice, be implemented using two base approaches, namely spinning or sleeping. The former allows for a fine-tuned control on the period of backing off, at the cost of spending energy similar to that of doing actual work, while the latter can save energy at the cost of coarse-grained control. Green-CM employs a hybrid approach that automates the choice between the spin-based and sleep-based implementations of the wait primitive, on the basis of the specified back-off period and the characteristics of the application workload.

The second contribution of Green-CM is Asymmetric Contention Management (ACM), namely the first CM policy designed to take advantage of the ability of modern processors to dynamically adjust their frequency and voltage (a mechanism known as Dynamic Voltage and Frequency Scaling or DVFS [39]). DVFS allows for adjusting dynamically the frequencies at which different cores operate: this allows both for reducing the energy

consumed by idle cores, and for increasing the frequency of active cores. ACM combines aggressive and conservative (i.e., exponentially vs linearly increasing) back-off policies, in order to promote the dynamic creation, at medium/high contention scenarios, of two sets of threads: *(i)* threads that are likely to be backing-off, allowing the corresponding processor to enter deep sleep states, and *(ii)* threads that spend most of their time executing transactions, and which can run at higher frequencies, thanks to DVFS.

To achieve energy efficiency across different architectures and wide variety of workloads, Green-CM employs a lightweight, on-line, hill-climbing-based, self-tuning mechanism to determine when to switch between different `wait` implementations and the degree of asymmetry of the ACM scheme. Compared with state of the art CMs, Green-CM was able to achieve 2.35× higher energy-delay product.

1.5 List of Publications

The following is the list of publications upon which Chapters 3, 4 and 5 are based. Permissions from the co-authors were obtained to include contents of these publications in this dissertation. Further, it was ensured that using the content of these publications does not represent a breach to their publishers' copyrights. A description of the thesis's author contribution is also summarized below.

I **Shady Issa**, Pascal Felber, Alexander Matveev, and Paolo Romano. Extending Hardware Transactional Memory Capacity via Rollback-Only Transactions and Suspend/Resume. In *LIPIcs 31st International Symposium on Distributed Computing*, volume 91 of *DISC '17*, pages 28:1–28:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik [Rank:A Conference]

II Pascal Felber, **Shady Issa**, Alexander Matveev, and Paolo Romano. Hardware read-write lock elision. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys '16, pages 34:1–34:15, New York, NY, USA, 2016. ACM [Rank:A Conference]

III **Shady Issa**, Pascal Felber, Alexander Matveev, and Paolo Romano. Extending Hardware Transactional Memory Capacity via Rollback-Only Transactions and Suspend/Resume. *Distributed Computing*, Submitted on January 15, 2018, currently under review

IV Pedro Raminhas, **Shady Issa**, and Paolo Romano. Enhancing efficiency of hybrid transactional memory via dynamic data partitioning schemes. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '18, pages 1–11, May 2018 [Rank:A Conference]

V **Shady Issa**, Paolo Romano, and Mats Brorsson. Green-CM: Energy efficient contention management for transactional memory. In *Proceedings of the 44th International Conference on Parallel Processing (ICPP)*, pages 550–559, Sept 2015 [Rank:A Conference]

The content of Chapter 3 is based on Papers I, II and III, while the content of Chapter 4 is based Paper IV and the content of Chapter 5 is based on Paper V.

For Papers I, III, IV and V the author of the thesis is the main contributor of the work. Specifically, he contributed to the conception and development of the idea. He was also a major contributor to the writing of the paper. For Paper II, the author of the thesis is a major contributor of this work. In particular, the author of this thesis has participated in the development of the algorithm.

For all the papers, the author of the thesis led and was responsible for the prototype implementations and carrying out the experimental evaluation using synthetic microbenchmarks and real-life complex applications.

1.6 Structure of this Document

The remainder of this document is structured as follows: Chapter 2 reviews state of the art of TM and the background necessary to frame and understand the contributions of this dissertation, which are detailed in the subsequent chapters. Specifically, Chapter 3 introduces POWER8-TM, then DMP-TM is explained in Chapter 4 and Green-CM in Chapter 5. Finally, Chapter 6 summarizes the work done in this thesis and highlights possible directions for future work.

Chapter 2

Background

The goal of this chapter is to review the state of the art of Transactional Memory (TM) systems, as well as to provide the necessary background to introduce the contributions of this dissertation. As the main purpose of TM is to provide an easy-to-use and efficient synchronization mechanism for parallel programming, the chapter starts with a brief overview of some of the main classic, i.e., not based on TM, synchronization mechanisms in Section 2.1. Section 2.2 focuses on TM, by first introducing the abstraction and the guarantees it offers to programmers, and then reviewing state of the art on TM systems. As discussed in the previous chapter, one common aspect of the contributions of this thesis is the use of on-line, self-tuning mechanisms to provide robust performance across wide variety of workloads; Section 2.3 overviews TM systems that also adopted self-tuning techniques. Finally, Section 2.4 describes several benchmarks that are popular within the TM community for evaluating TM systems and that are going to be used to evaluate the solutions presented later on in this dissertation.

2.1 Conventional Synchronization Mechanisms

2.1.1 Mutual Exclusion

The traditional technique for ensuring correct synchronization in parallel programming is based on the notion of mutual exclusion and on the abstraction of mutex locks. As the name suggest, the idea at the basis of mutual exclusion is to ensure that programmer-defined code segments, referred to as critical sections, are never executed concurrently [45]. A simple, yet powerful, extension of the mutual exclusion lock, called read-write lock, is based on the idea of enabling concurrency between read-only critical sections, which, not modifying shared data, can be executed concurrently in a safe way [46].

The problem of building efficient implementations of mutex and read-write locks has been long studied in the literature. An overview of some of the most popular implementations of these synchronization mechanisms is provided in the following.

Mutexes

The most common way of ensuring mutual exclusion is the use of locks or mutexes. When using mutexes, in order for thread to start executing a critical section, it must acquire the mutex before beginning to execute the critical section and then release the mutex after it finishes. A mutex can only be acquired by one thread and while the mutex is held by a thread, any other thread trying to acquire it will be blocked, and thus prevented from entering a critical section.

A mutex can be implemented either using software or using atomic instructions that are implemented in hardware (e.g.: fetch-and-increment, compare-and-swap, test-and-set, etc.). Hardware-based mutex implementations are known to be faster and easier to prove correct, thanks to the atomic support of the hardware [47]. The literature on lock algorithms, possibly using hardware mechanism, is quite vast. In the following, some of the most popular mutex lock implementations are overviewed.

The most basic implementation is the Test and Set (TS) mutex [48] where a thread that intends to acquire the mutex repeats executing the atomic test-and-set instruction until it successfully changes the value of the mutex from free to acquired. The thread can release a TS mutex by simply setting its value to free. Test, Test and Set (TTS) mutex [49] introduces a simple, yet effective, modification to the TS mutex. Instead of blindly executing the expensive test-and-set instruction, the acquiring thread would only issue it after a plain read operation indicating that the mutex is free. The Ticket mutex [50] is similar to Take-a-Number systems used at butcher shops. The Ticket mutex uses two counters, called *ticket* and *now serving*. When a threads requests to acquire the mutex, it atomically fetches and increments *ticket* and waits for *now serving* to reach that value. Upon releasing the lock, the thread increments the *now serving* counter. Compared to the TS and TTS mutexes, the Ticket mutex has stronger fairness guarantees, as threads are served in FIFO order. Further, since the Ticket mutex allows only one thread to enter the critical section at a time, it avoids the "*thundering herd*" problem [51] that arises in TS and TTS when multiple threads attempt to acquire the mutex simultaneously. The MCS mutex [48] uses compare-and-swap instruction to implement a FIFO queue where each thread requesting to acquire the mutex enqueues itself in the queue. While in the queue, each thread keeps spinning on a local variable that changes only when the thread becomes the head of the queue, only then it can access the critical section. When a thread releases the mutex, it dequeues itself and changes the local variable of the next thread in the queue. In addition to providing FIFO properties, similar to the Ticket mutex, the MCS mutex further reduces cache traffic since each thread spins on a local variable while the lock is held busy.

Regardless of how a mutex is implemented, the main disadvantage of using mutexes is that they can hinder performance by restricting parallelism. If not designed carefully, programs using mutexes can, indeed, achieve worse performance than in the case of sequential execution [52]. One key aspect that affects the efficiency of mutex-based programs is the granularity of critical sections. To achieve higher scalability, programmers can use different mutexes to protect different shared data, hence allowing more room for parallelism.

The use of fine-grained locking schemes brings both advantages and disadvantages. The main advantage is the performance gains, the more fine grained they are the more the parallelism they allow and thus the better the performance. However, this gain comes at a cost, which is the complexity of designing and developing such software. Deadlocks are one of the major difficulties programmers are faced with when using fine-grained locking schemes. A deadlock happens when two or more threads are waiting for each other to release a mutex that they are holding [53], thus suffer of liveness issues. Debugging and detecting deadlocks is not trivial and is well known to be a complex and time-consuming task [52]. For instance, attempts to solve a deadlock may result into livelocks, a state in which threads do not mutually block but still fail to make any actual forward progress.

Read-write Locks

Read-write Lock (RWL) is a synchronization mechanism that is directed towards read dominated workloads applications that have majority of critical sections that only read shared data, without modifying it [46]. The idea behind RWL is to support concurrent readers, while allowing the execution of only one critical section that modifies shared data at a time.

Several works addressed the problem of devising efficient implementations of the RWL abstraction. The most famous is probably the implementation provided by the pthread's library, which uses two counters to synchronize both readers and writers [54]. The pthread's implementation has an internal mutex that is used to synchronize the changes to these counters, and the values of the counters are used to ensure fairness between readers and writers.

Another RWL implementation is the big reader lock (BRLock) [55] which was part of the Linux kernel at one point [56]. The key idea behind BRLock is to trade write throughput for read throughput. A thread acquiring BRLock in read mode will only acquire one private mutex, whereas acquiring BRLock in write mode entails acquiring all private mutexes of all running threads.

Recently, Liu et. al [57] introduced passive reader writer lock (PRWL), a synchronization mechanism that tries to reduce the cost imposed by most reader writer locks on the writer mode. The idea behind PRWL is a version based consensus protocol between readers and writers. Writers increment the lock version and wait for readers to signal they have read the latest version. This is designed for total store order systems where an upper bound on the memory staleness can be guaranteed, i.e., the time taken by readers to see the latest version without a memory barrier.

2.1.2 Read-Copy-Update

Read-Copy-Update (RCU) [36] is a popular synchronization mechanism that targets read-dominated workloads. Unlike RWL, with RCU, a read-only critical section does not need to acquire any mutex, it just flags itself, using a memory barrier, at the beginning and end of the critical section. To ensure correctness, a writer modifying shared data, would create

a copy of the data and apply the updates to the copy. Readers that existed prior to the write continue to access the older, unmodified data, while new readers get to witness the updates. Only when all readers that existed before the writer have completed their critical sections, the unmodified data is discarded.

Although RCU can provide significant performance gains, it comes with a high cost in terms of usability. To support RCU, programs must be written in a way such that creating copies of shared data is feasible. Further, duplicating and discarding the copies must maintain correct pointers to other referenced data. This limits the applicability of RCU and is arguably the main reason why only a relatively small number of RCU-based data structures have been proposed in the literature [58–60].

2.2 Transactional Memory

Transactional memory (TM) is a synchronization paradigm for parallel programming that aims at avoiding the complex problems that arise when using fine-grained locking schemes (see Section 2.1.1), while achieving comparable, or possibly even higher, performance. The ease of use of TM stems from the simplicity and familiarity of the key abstraction that it provides, i.e., which hides all the complexities of handling data races from the programmer.

TM can be implemented as a software runtime, in hardware or as a hybrid combination thereof. In all cases, it provides the same abstraction and guarantees to the programmer.

Listing 2.1: Sample C++ code using TM abstraction to checkout a shopping bag

```
int buy(item i){
        if (stock[i.type] >= i.quantity){
                stock[i.type] -= i.quantity;
                return 1;
        }
        return 0;
}

int checkout(shoppingBag bag){
        int result = 1;
        __transaction_atomic{
                for(auto item: bag){
                        result &= buy(item);
                }
        }
        return result;
}
```

2.2.1 Abstraction

The abstraction provided by TM allows programmers to devise their code into blocks or transactions, that are to be executed atomically. The underlying TM implementation will execute transactions concurrently, but still offer the illusion of sequential executions, hence significantly simplifying the reasoning on correctness of concurrent programs.

Listing 2.1 shows a sample C++ code for a function that checks out a shopping bag. It performs the *buy* operation on each item in the bag. To avoid overselling items, the *buy* function will check for available inventory. As several customers may be checking out at the same time, using a single global lock to protect access to available stock would only allow a single customer a time. Fine-grained locking can allow more concurrent customers, but at the cost of an increased complexity (e.g., risk of deadlocks).

With TM, programmers just need to define the block as a transaction using the `__transaction_atomic` construct. The compiler will then plug in the calls to transaction *begin* and *commit* according to the TM implementation. Further, in case the selected TM implementation requires software instrumentation to track transactional accesses to shared data, the compiler will automatically replace these accesses with the respective functions from the TM API. Listing 2.2 shows the *buy* function with instrumented accesses to shared data.

Listing 2.2: Code with instrumented accesses to shared data

```
int buy(item i){
        if (TM_SHARED_READ(stock[i.type]) >= i.quantity){
                TM_SHARED_WRITE(stock[i.type], \
                TM_SHARED_READ(stock[i.type]) - i.quantity);
                return 1;
        }
        return 0;
}
```

2.2.2 Guarantees

Guerraoui and Kapałka [61] coined the term *opacity* to formalize the correctness guarantee that should be ensured by a TM implementation. Roughly speaking, opacity requires both committed and aborted transactions to be explicable by considering an equivalent sequential execution that respects the real-time order of the original, sequential, execution. This is in contrast with *strict serializability* [62], which provides the same guarantees but only for committed transactions, while providing no restrictions on the states (also called snapshots) observable by aborted transactions — which could potentially be exposed to arbitrary anomalies. Opacity ensures that changes produced by committed transactions appear to happen atomically, while changes within aborted transactions are discarded and hidden from concurrent transactions to avoid any side effects. Therefore, with opacity, the read-set of a TM transaction is always consistent until it either commits or aborts. This is necessary when using transactions in generic programs written using low-level programming languages and directly manipulating shared memory — a non-constrained domain, unlike databases where the detrimental effects effects of reading inconsistent values can be easily contained using sand-boxing techniques [63].

2.2.3 Design Choices

The large number of existing TM implementations can be coarsely classified according to the design choices they adopt for what concerns two key aspects: when to detect conflicts and which granularity to use to track transactional accesses.

Timing of Conflict Detection

A conflict is said to occur when concurrent transactions access the same shared data, and at least one of them tries to update it. Conflicts can be detected either eagerly, or lazily. When conflicts are detected eagerly, transactions are aborted and rolled back once the conflict occurs. In contrast, in lazy conflict detection, conflicts are only detected at commit time. Thus, lazy conflict detection is more optimistic as it allows transactions to execute regardless of whether they conflict or not. This is beneficial in low contention workloads where more parallelism can be achieved by doing so. It is worth noting here that the update of active (i.e., non-committed) transactions are hidden from each other. However, in high contention scenarios, a more pessimistic concurrency control mechanism is beneficial, hence eager conflict detection [63]. Various works [7, 8] have shown that, in practice, no one size fits all, and different designs are needed for different workloads.

Granularity of Conflict Detection

Data accesses within a transaction need to be tracked in order to detect conflicts between concurrent transactions. The granularity of tracking can range from word based to object based. This represents a trade-off between accuracy and the overheads of tracking. Similar to conflict detection, there seems to exist no one size fits all solution [7]: fine grained tracking allows more parallelism at the cost of increased overheads while coarse grained tracking suffers from aliasing issues, which can lead to spurious aborts and ultimately hinder performance.

2.2.4 Software Transactional Memory

Despite the fact that TM was originally conceived as supported via a hardware extension of cache coherence, the first concrete implementations of TM systems were as a software runtime. This was arguably due to the flexibility and portability that the software provides. Software Transactional Memory (STM) provides programmers with an API to demarcate blocks of code as transactions. Shared data access within a transaction have to be issued via the STM API - which can be automated using compiler support. During the past decades, STM went through a thorough research that yielded a large number of algorithms that implement STM efficiently for different workloads. In the following, some of the most popular STM implementations, representative of different design choices, are described.

TL2

TL2 [6], uses a versioned write lock (also known as ownership record, or *orec*) for each tracked item, whether this is a memory word or a shared object. It also uses a single global clock to give timestamps to tracked items upon each update. When a transaction starts in TL2, it atomically reads and caches locally the value of the global clock. Upon each read or write, it makes sure that the value of the *orec* of the item, which is being read or written, is less or equal to the value of the cached clock it read at the beginning of the transaction and that the write lock is free. Otherwise, the transaction must abort. When the transaction requests to commit, it has to acquire the *orecs* of all the items in the write-set. Next, the transaction atomically increments the global clock using a *compare-and-swap* operation, then re-validates the read set. The new values of the write-set are then buffered to memory before releasing the *orecs* while updating their versions to the new value of the global clock.

TinySTM

TinySTM [7] is based on a single version, word based variant of Lazy Snapshot Algorithm (LSA) [64]. Generally speaking, TinySTM is similar to TL2. One main difference is that TinySTM uses encounter-time locking instead of commit-time locking. This makes TinySTM avoid performing useless work, since conflicts will be detected eagerly, and enables handling read after write conflicts without always aborting the reading transaction.

In LSA, when a transaction is performing a read, and the version of the *orecs* of the item being read is higher than the clock read at the beginning of the transaction, the transaction can try to extend its snapshot instead of aborting. This is done by checking that all items in the read-set are still valid and are not locked by any other transaction. If this is the case, the transaction can safely read the value and update the value of its clock to the new clock value, hence extending its snapshot.

SwissTM

SwissTM [8] uses a mix of the techniques employed by TL2 and TinySTM, where it uses commit-time locking for read write conflicts and encounter-time locking for write write conflicts. It also augments the concurrency control algorithm with a contention management scheme that tries to favor older transactions without starving newer ones. The main intuition behind SwissTM is to minimize doing useless work, or wasting work done.

NOrec

Unlike all the previous STM algorithms, NOrec [9] does not have an *orec* for each tracked item, hence its name (No + *orec*). In contrast, NOrec has only a single global clock that it uses to ensure correct execution of transactions. This comes with the major benefit of minimizing the overheads of tracking the read and write sets. However, to ensure safe execution, the read-set must be validated whenever another transaction commits. Further, since there are no *orecs*, the only way to do this is by comparing the values in the read-set

to the values in memory. This makes NOrec very efficient at low number of threads, i.e., when there is not a high number of concurrently committing transactions.

When a transaction starts, it reads the global clock and upon each read it checks the current value of the clock. In case the value of the clock has changed, it has to re-validate its read-set. Only if all the values have not changed the transaction can continue. Otherwise, it must abort. When the transaction wants to commit, it has to acquire a global lock, which allows only one transaction to commit at a time. Next, it validates its read-set, applies the write-set to memory, increments the global clock and then releases the lock.

2.2.5 Hardware Transactional Memory

As mentioned earlier, TM was initially proposed as a hardware mechanism, with an implementation based on the idea of extending the cache coherence protocol already in use by existing parallel processors. The main advantage of implementing TM in hardware is to address the problem of the costly tracking the read and write sets in software. Several works were conducted on how to efficiently implement Hardware Transactional Memory (HTM) [65–68]. However, only 20 years after its initial proposal, a hardware implementation of TM was first available in commercial processors. Before that, there were several unsuccessful attempts such as the Rock Processor by Sun [69], which never made it to the market, and the AMD Advanced Synchronization Facilities (ASF) [18], which remained a proposal, but was never really implemented in real hardware.

One major drawback of all HTM proposals, including the ones available in real life processors nowadays, is their best effort nature. This means that transactions may never be able to commit successfully in hardware even in absence of concurrency, due to reasons such as the size of the transaction or execution of prohibited instructions. The following describes two of the current realizations of HTM in commodity hardware that are provided by the two major processor manufacturers: Intel and IBM.

HTM Implementation in Intel Processors

In 2013, Intel released the Haswell processor, which was their first commercial processor to include support of HTM [19]. Intel's HTM implementation was accompanied by the release of an extension of the instruction set, named Transactional Synchronization Extensions (TSX). TSX is a set of new instructions, which allows programmers (or, more typically, compilers) to mark the beginning and end of transactions. The hardware then guarantees the atomicity of the transactions while hiding their updates until they commit successfully. Intel implemented HTM in the cache coherence protocol with conflict detection granularity of a single cache line. Empirical studies found out that writes are tracked in L1 cache, and it is less clear how reads are tracked, as the available read capacity is larger than L2's capacity but smaller than L3's — which has led some authors to conjecture the use of some probabilistic data structure like bloom filters [70, 71].

TSX provided two interfaces for programmers to use HTM, one is Hardware Lock Elision (HLE), which performs hardware lock elision, and the other is Restricted Transactional

Table 2.1: List of instructions provided in Intel TSX

Command	Description
`xacquire`	acquire a lock in HLE mode
`xrelease`	release a lock in HLE mode
`xbegin`	start a RTM transaction
`xtest`	check if inside RTM transaction
`xabort`	abort RTM transaction
`xend`	commit RTM transactions

Memory (RTM). The main difference is that HLE allows for automatically replacing critical sections in legacy applications with speculative transactions, whose retry logic is fully managed in hardware. Conversely, RTM allows programmers to specify a custom transaction retry logic in case the transaction fails to commit successfully in hardware (e.g., a single global lock).

When programs request to start a transaction, a *started* code is placed in the, so called, status buffer. If, later, the transaction aborts, the program counter jumps back to just after the instruction used to begin the transaction. Hence, in order to distinguish whether a transaction has just started, or has undergone an abort, programs must test the status code returned after beginning the transaction.

HTM Implementation in IBM POWER8 Processors

This section provides background on POWER8's HTM system. Analogously to TSX, POWER8 has also new instructions to mark the beginning and end of transactions. In addition to that, it also supports Suspend/Resume (S/R) instructions, which allow programmers to execute non-transactional code within a transaction.

Table 2.2: List of instructions provided in POWER8 HTM

Command	Description
`TM_begin(_rot)`	start a HTM (ROT) transaction
`TM_abort`	abort HTM transaction
`TM_named_abort`	abort HTM transaction with specific code
`TM_suspend`	escape the transactional context
`TM_resume`	return to the transactional context
`TM_end`	commit HTM transactions

POWER8 detects conflicts with granularity of a cache line similar to Intel's HTM. However, instead of relying solely on caches to track accesses from within transactions; POWER8 HTM uses a per hardware thread buffer, called TMCAM. In practice, the trans-

action capacity is bound by TMCAM's 8KB size, which stores the addresses of up to 64 distinct cache lines read or written within the transaction [72].

In addition to plain HTM transactions, POWER8 also supports a special type of transactions, called Rollback-Only Transactions (ROT). The main difference being that, in ROTs, only the writes are tracked in the TMCAM, giving virtually infinite read-set capacity. As such, ROTs cannot be used to regulate concurrent execution of generic transactions, but are rather conceived as a mechanism aimed at ensuring failure-atomicity in single threaded applications [21]. Both HTM transactions and ROTs detect conflict eagerly, i.e., they are aborted as soon as they incur a conflict.

Table 2.3: List of abort causes in POWER8 HTM

Abort type	Description
`HTM/ROT trans conflict`	conflict with another transaction
`HTM/ROT non-trans conflict`	conflict with non-transactional code
`HTM/ROT self conflict`	conflict with S/R block
`HTM/ROT buffer overflow`	capacity exception
`HTM/ROT user`	explicit abort call

2.2.6 Hybrid Transactional Memory

As mentioned earlier, existing HTM implementations adopt a bet effort approach and, in order to guarantee liveness and forward progress, HTM must be complemented by a software fallback. This fallback can be as simple as a global lock or as complex as a STM. In the latter case, the resulting system is often referred to as Hybrid Transactional Memory (HyTM) [22, 23]. The main challenge of designing HyTM is how to ensure correctness while executing transactions concurrently in hardware and in the fallback software path without incurring significant overheads [29] in both the hardware and software paths — sometimes referred to as fast path and slow path. In the following, an overview of some of the most representative HyTM implementations is provided.

Hybrid NOrec

Hybrid NOrec [24] or HyNOrec is one of the most popular implementation of HyTM. This is due to its simplicity and performance as compared to other HyTM proposals. As the name suggests, HyNOrec uses the NOrec STM as a fallback. Since NOrec uses only a single global lock and only one update transaction is allowed to commit a time, minimal changes are required to the fast path in order to ensure correctness.

When a transaction starts in HTM, it has to read the current value of NOrec's global lock — an operation that is often referred to as lock subscription in the literature, e.g. [73]. This is to ensure that when STM update transactions start their commit phase (during which they have to acquire the global lock) they abort any concurrent HTM transactions

to prevent them from reading inconsistent states during the write back phase of the STM transaction. Finally, before a transaction commits in hardware, it must increment the global clock to inform the concurrent STM transactions that it has committed — and may have possibly updated data already accessed by the STM.

Several optimizations were proposed to enhance the performance on HyNOrec. A first optimization consists in using a per thread counter to be incremented upon HTM commit instead of a global clock. This can avoid spurious aborts caused by the commit of concurrent HTM transactions; however, it introduces larger overheads for STM transactions, which need to check all the per thread counters upon each read. An alternative approach, also aiming at avoiding spurious aborts between HTM transactions, consists in incrementing the global clock non-transactionally [25]. However, none of the existing HTM implementations support the execution of individual non-transactional memory accesses from within the context of a transaction. IBM's POWER8 is only architecture that allows to approximate this behavior, via its mechanism to S/R transactions, which, however, has non-negligible costs [21].

Invyswell

After the introduction of Intel Haswell, Calciu et. al [27] proposed Invyswell as a HyTM solution for Intel's HTM. Invyswell has five different types of transactions: 1) STM transactions that are based on InvalSTM [74], 2) HTM transactions with software tracking of read and write sets, 3) plain HTM transactions, 4) single global lock and 5) irrevocable software transactions.

In order to ensure correctness, not all types are allowed to execute concurrently. HTM transactions without software tracking are only allowed to execute concurrently with other HTM transactions, preventing concurrency with any other type of transaction. Invyswell uses some thresholds to determine when to switch from one type of transaction to another. It also relies on heuristics to enable and disable different types according to nature of the workloads.

Reduced HyNOrec

Reduced HyNOrec [28] or RHyNOrec was introduced as an improved version of HyNOrec. The main idea behind RHyNOrec is to replace the *slow path* with a new path, called *mixed path*, that tries to execute as much as possible of the slow path inside hardware transactions.

When a mixed path transaction begins, it starts a "*prefix*" hardware transactions executing as much reads as possible until it either reaches the first write, or consumes all the available transactional capacity. It then reads the global clock and tries to commit the "*prefix*" hardware transaction. After that, it executes normally according to NOrec's logic, using the global clock it read at the end of the '*prefix*" transaction as the starting value. When it reaches the commit phase. In the commit phase it acquires the lock, then starts a "*postfix*" hardware transaction to write back the write-set atomically.

If no transaction executes in the slow path, transactions executing in fast path do not need to read the value of the clock at the beginning of the transaction as the STM writes are executed using a hardware transaction. This eliminates false conflicts that arise in HyNOrec when HTM transactions are aborted upon the commit of any STM transaction. However, there are no guarantees that the hardware transactions in the mixed path will ever succeed, due to the best effort nature of HTM. To this end, RHyNOrec makes use of some heuristics to when to fallback to the slow path and execute similarly to HyNOrec

Hybrid-LSA

Riegel et. al [25] proposed Hybrid LSA (HyLSA) based on AMD's ASF (see Section 2.2.5), to use highly scalable LSA-based STM systems, such as TinySTM, as a fallback for HTM. Hybrid-LSA incurs high instrumentation costs in its fast path upon each read, write and the commit phase to ensure correctness between concurrent HTM and STM. The basic intuition of HyLSA is to instrument accesses to shared data from within the fast path and make them *orec*-aware. For a successful HTM read, the *orec* has to be found free. Analogously, when writing, the HTM must acquire the *orec*, in order to make sure that it will be aborted if a concurrent STM transaction accesses the respective same *orec*. Finally, upon commit, the HTM must update the global clock and the *orecs* of its write-set, similar to TinySTM.

HyLSA relies on the availability of non-speculative and *prefetchw* operations, which were defined in the ASF's specification, but are not available on any current HTM implementation. The only approximation for the former, which is to perform non-speculative operations between S/R calls available on the POWER8 processor. This may not be practical as these operations are to be performed upon each read access, hence incurring a high cost with current HTM implementations. Although this operation is not directly supported by existing HTM implementations, it can in practice be emulated by reading and writing back the value currently stored in the target memory position.

Hybrid-TL2

Matveev and Shavit [26] proposed an algorithm to employ another *orec*-based STM, TL2, as a fallback for HTM, without the need of instrumenting read accesses to shared data. With Hybrid TL2 (HyTL2), only writes issued from within transactions in the fast path are instrumented to increment the respective *orec*. This notifies concurrent slow path transactions of updates performed in the fast path. To protect HTM transactions from witnessing inconsistent values, STM transactions need to acquire a global lock, that HTM subscribes to, at the commit time.

2.2.7 Contention Management and Scheduling

Most TM designs follow an optimistic design in order to favor scalability in uncontended workloads and avoid deadlocks. The consequence of such a design choice is that TM implementations are likely to incur high abort rates in contention prone workloads with

detrimental effects on the efficiency of the system. The TM literature has investigated a large number of techniques that tackle precisely this issue and that can be seen as complementary and orthogonal to concurrency control scheme employed by the TM system. The existing solutions in this space are typically classified as either Contention Management (CM) or scheduling techniques.

CM mechanisms adopt a reactive policy, which is responsible to determine what to do when conflicts do arise. Basically, when a conflict is detected, the CM is consulted on which conflicting transaction to abort and when to restart it. Conversely, scheduling techniques act proactively and aim at avoiding conflicts by preventing the concurrent execution of transactions that are likely to generate conflicts.

Contention Management Policies

In the TM literature, a large number of CM schemes have been proposed [75–78]. Below, some of the most popular CM techniques are overviewed:

- The *polite* CM backs off aborted transactions for exponentially increasing waiting periods after each failed attempt.

- In both the *karma* and the *polka* CMs, an attacking transaction, i.e., a transaction that generates a conflict with some other concurrent transaction, makes a number of attempts equal to the difference among priorities (calculated from the timestamps) of both transactions. In karma, there is constant back-off between each attempt, this back-off grows exponentially in polka.

- With the *greedy* CM, if the transaction that is targeted by the conflict, or the victim transaction, is waiting or has lower priority than the attacking transaction, then the attacking transaction aborts it; otherwise, the attacking transaction waits, until the victim either commits, aborts or starts waiting.

- The *timestamp* CM forces the attacking transaction, in case it is not older than the victim, to wait for a predetermined number of time intervals. After attempting half the number of intervals, the CM flags the victim as possibly defunct. After attempting the full number of intervals, if the victim has the defunct flag set, the victim is aborted. Meanwhile, if the victim transaction performs any transaction-related operation, its defunct flag is reset.

- With the *eruption* CM, the attacking transaction waits exponentially for the victim transaction if it has higher priority and increases its priority; otherwise the victim transaction is aborted.

- Both, the *suicide* and the *aggressive* CMs abort the attacking and the victim transactions respectively.

- The *two-phase* CM uses *suicide* for read only and short transactions and *greedy* for more complex transactions. This is the scheme employed by SwissTM that was described in Section 2.2.4.

As for HTM systems, the fact that existing HTM implementations automatically decide which transaction to abort upon a conflict and does not expose information such as the read and write-sets and which transaction caused the conflict, makes most CMs not applicable. From the above list only, polite can be used for HTM contention management.

Schedulers

CMs react after a conflict takes place to try to minimize its effect in the future, whereas schedulers act proactively to try to avoid conflicts in the first place. There also exists a large amount of works that addressed the problem of scheduling in the context of TM systems. Some of the most notable solutions in this space are briefly overviewed next:

- *Seer* [79] uses a probabilistic model to detect conflicts in HTM, and accordingly force the forces serialization of transactions that are predicted as prone to conflict.
- *Steal-On-Abort* (SOA) [80] and *CAR-STM* [81] assume a queue for each core. Upon abort of a transaction, it is rescheduled to the queue of the core that executes the other conflicting transaction.
- *ProPS* [82] used abort events to calculate contention probabilities between transactions. When a transaction starts it may be forced to wait if it is ought to conflict with a running transaction with a probability higher than a certain threshold.
- The *shrink* scheduler [83] is similar to ProPS, but in addition to abort events it uses the read and write-sets to calculate the conflict probability.
- Both *SER* [84] and *TxLinux* [85] modified the Linux scheduler to be transaction aware.
- *ATS* [86] serializes all transactions using a single global lock when a contention factor, that is calculated as a function of commits and aborts, exceeds a certain threshold.

Similar to CMs, not all schedulers can be used with HTM. Indeed, in the above list, only ATS and Seer can be used with HTM. SOA and TxLinux were proposed as HTM schedulers. However, they were designed for hypothetical HTM designs that provided features not available in current real life HTM implementations.

2.3 Self-tuning for Transactional Memory

"No size fits all" is probably the most repeated phrase when discussing the design choices of existing TM systems. Different workloads with different characteristics require different configurations. Indeed, several studies [70, 72, 87, 88] have shown that, given the large

number of factors affecting the workload characteristics of TM applications, there exists no single TM implementation that excels across all workloads. This observation has motivated several proposals that leverage self-tuning mechanisms to adapt various internal mechanisms or configuration parameters of TM systems to best fit the workload characteristics.

Tuner [89] tackles the problem of deciding when to revert to the fallback path in case transactions fail repeatedly to execute in hardware. To pursue this goal, Tuner relies on two reinforcement learning techniques: the Upper Confidence Bounds (UCB) scheme [90], which Tuner uses to identify the optimal retry policy before activating the pessimistic fallback path upon a capacity exception; and a (probabilistic variant of) hill-climbing [91], which is used to determine the retry budget.

It is well known [92] that, in many TM workloads, the highest performance is achieved at a number of threads lower than the maximum degree of parallelism allowed by the underlying hardware platform. This motivated the investigation of automatic techniques aim at identifying the optimal level of parallelism. The various works published in this area can be classified into three categories depending on whether they rely on analytical modeling, offline learners and on-line learners. Offline learners use machine learning techniques to build black-box models, trained used data-sets gathered prior to deploying the actual TM application [93, 94]. Solutions based on analytical models, conversely, tend to rely on white-box models that depart from a (possibly approximate) knowledge of the internal dynamics of the TM system [95, 96]. Finally, on-line learners use black box techniques, which are designed to exploit on-line feedback on system's performance when using different system configurations [92, 97, 98].

Unlike the previously mentioned solutions, which focus on tuning a single aspect of TM systems, ProteusTM [99] explores a multidimensional space including the TM implementation, the degree of parallelism and then contention manager. To achieve this, ProteusTM relies on techniques used in recommender systems [100], which entails offline training to build performance predictors and on-line mechanisms to detect new workloads and predict their optimal configuration.

2.4 Benchmarks for Transactional Memory

This section describes some of the most popular benchmarks used in the literature to evaluate TM systems. These benchmarks shall be used, in the following chapters, to assess the efficiency of the various contributions of this dissertation.

Concurrent data structures. TM systems are often evaluated using simple benchmarks based on concurrent data structures, like hashmap, linkedlist and redblack trees. These benchmarks typically have three types of operations: lookup, insert and delete. The ratio of each operation in the workloads together with the structure and size of the datastructure allow for controlling factors such as degree of contention and size of transactions.

STAMP. This is a popular benchmark suite for TM systems, which contains seven different applications that emulate real world problems [101]. The different applications have different characteristics in terms of size of transactions and degree of contention. The input parameters allow to further tune the behavior of applications to stress certain aspects.

STMBench7. Another popular TM benchmark that has a large number of different transactions that operate on a datastructure of graphs and indexes, which mimic CAD applications [102]. Input parameters allow to exert control over length of graph traversals, whether or not to allow structural modifications and the percentage of read only operations.

TPC-C is well known benchmark in the database community [103]. It represents a wholesale supplier benchmark for relational databases, it consists of 5 different types of transactions, two of which are read-only. This dissertation will make use of a version of TPC-C that was ported to operate on in-memory database [104, 105] and, straightforwardly, adapted to support TM.

Memcached is popular in memory key-value store [106]. Ruan et. al [107] ported the lock based code of Memcached to a transactionalized version that uses the GCC TM API which follows the C++ TM specifications [4], and, thus, can allow for plugging in custom algorithms easily.

Chapter 3

POWER8-TM

This chapter presents POWER8-TM (P8TM), a novel TM that exploits two specific features of specific features of the HTM implementation of IBM POWER8 processors, namely Suspend/Resume (S/R) and Rollback-Only Transactions (ROTs), in order to overcome (or at least mitigate) one of the key limitations stemming from the best-effort nature of existing HTM system, namely the inability to execute transactions whose working sets exceed the capacity of CPU caches. Some passages in this chapter have been quoted verbatim from [40, 41].

3.1 Problem

Over the last few years, hardware supports for transactional memory (TM) have been integrated in several mainstream commercial processors employed in a variety of computing platforms, ranging from commodity systems (Intel's Haswell [19]), to servers (IBM's POWER [14]) and super computers (IBM zEC12 [16]).

Existing hardware implementations share various architectural choices, although they do come in different flavors [15, 16, 19, 108]. The key common trait of current HTM systems is their best effort nature: current implementations maintain transactional metadata (e.g., memory addresses read/written by a transaction) in the processor's cache or special hardware buffers. Due to the inherently limited nature of processor caches, current HTM implementations impose stringent limitations on the number of memory accesses that can be performed within a transaction, hence providing no progress guarantee even for transactions that run in absence of concurrency. As such, HTM requires a fallback synchronization mechanism (also called *fallback path*), which is typically implemented via a pessimistic scheme based on a single global lock.

An important conclusion reached by several studies [70, 72, 88] is that HTM's performance excels with workloads that fit the hardware capacity limitations. Unfortunately, though, HTM's performance and scalability can be severely hampered in workloads that contain even a small percentage of transactions that do exceed the hardware's capacity. This is due to the need to execute such transactions using a sequential fallback mechanism

based on a single global lock (SGL), which causes the immediate abort of any concurrent hardware transactions and prevents any form of parallelism.

3.2 Overview

To tackle the capacity limitation of HTM systems, P8TM incorporates several techniques, which operate in synergy to expand the effective capacity available for update and read-only transactions, as well as to preserve efficiency in adverse workloads.

Uninstrumented read-only transactions (UROs). P8TM executes read-only transactions outside of the scope of hardware transactions, hence sparing them from the spurious aborts and capacity limitations that affect HTM, while still allowing them to execute concurrency with update transactions. This result is achieved by exploiting the S/R mechanism of POWER8 to implement a RCU-like quiescence scheme where updates are not made visible until already active readers have committed (see Section 2.1.2). The quiescence shelters UROs from observing inconsistent snapshots that reflect the commit events of concurrent update transactions. A detailed description of how UROs are managed by P8TM is provided in Section 3.3.1

ROT-based update transactions. In typical TM workloads, the read/write ratio tends to follow the 80/20 rule, i.e., transactified methods tend to have large read-sets and much smaller write sets [109]. This observation led to development of a novel concurrency control scheme based on a novel hardware-aware software design: it combines the hardware-based ROT abstraction—which tracks only transactions' write sets, but not their read-sets, and, as such, does not guarantee isolation—with software based techniques aimed to preserve correctness in presence of concurrently executing ROTs, UROs, and plain HTM transactions. Specifically, P8TM relies on a novel mechanism, called Touch-To-Validate (T2V), to execute concurrent ROTs safely. T2V relies on a lightweight software instrumentation of reads within ROTs and a hardware aided validation mechanism of the read-set during the commit phase. The T2V algorithm is detailed in Section 3.3.2.

HTM-friendly (software-based) read-set tracking. A key challenge that had to be tackled while designing P8TM was developing a "HTM-friendly" software-based read-set tracking mechanism. In fact, all the memory writes issued from within a ROT, including those needed to track in software the ROT read-set, are transparently tracked in hardware. As such, the read-set tracking mechanism can consume cache capacity that could be otherwise used to accommodate application-level writes issued from within a ROT. Section 3.5 presents two read-set tracking mechanisms that explore different trade-offs between space and time efficiency.

Self-tuning. In order to ensure robust performance in a broad range of workloads, P8TM integrates a lightweight reinforcement learning mechanism (based on the UCB algorithm

[110]) that automates the decision of whether: *(i)* to use upfront ROTs and UROs, avoiding at all to use HTM; *(ii)* to first attempt transactions in HTM, and then fallback to ROTs/UROs in case of capacity exceptions; or *(iii)* to even completely switch off ROTs/UROs, using only HTM. The self-tuning mechanisms of P8TM are detailed in Section 3.6.

Finally, Section 3.7 shows the results of an extensive evaluation study of P8TM that encompasses synthetic micro-benchmarks and complex real-life applications from the STAMP suite [101] and the popular TPC-C benchmark [103]. The results of the study show that P8TM can achieve up $\sim$7$\times$ throughput gains with respect to plain HTM and extend its capacity by more than one order of magnitude, while remaining competitive even in unfavorable workloads.

3.3 Description

The key challenge in designing execution paths that can run concurrently with HTM is efficiency: it is hard to provide a software-based path that executes concurrently with the HTM path, while preserving correctness and speed. The main problem is that the protocol must make the hardware aware of concurrent software memory reads and writes, which requires to introduce expensive tracking mechanisms in the HTM path.

P8TM tackles this issue by exploiting two unique features of the IBM POWER8 architecture: *(i)* S/R for hardware transactions, and *(ii)* ROTs. P8TM combines these new hardware features with a RCU-like quiescence scheme in a way that avoids the need to track reads in hardware. This can in particular reduce the likelihood of capacity aborts that would otherwise affect transactions that perform a large number of reads.

The key idea is to provide two novel execution paths alongside the HTM path: *(i)* a, so called, *URO path*, which executes read-only transactions without any instrumentation, and *(ii)* a, so called, *ROT path*, which executes update transactions that do not fit in HTM as ROTs.

HTM transactions and ROTs exploit the speculative hardware support to hide writes from concurrent reads. This allows coping with read-write conflicts that occur during ROTs/UROs, but it does not cover read-write conflicts that occur after the commit of an update transaction. For this purpose, before an update transaction (running either as a HTM transaction or a ROT) commits, it first suspends itself and then executes a quiescence mechanism that waits for the completion of currently executing ROTs/UROs. In addition to that, in case the committing update transaction is enclosed in a ROT, it further executes an original *touch-based validation* step, which is described later, before resuming and committing. This process of "suspending and waiting" ensures that the writes of an update transaction will be committed only if they do not target/overwrite any memory location that was previously read by any concurrent ROT/URO.

3.3.1 Uninstrumented Read-Only Transactions

P8TM exploits the S/R mechanism to execute read-only transactions without resorting to the use of hardware transactions or performing instrumentation of read operations on shared

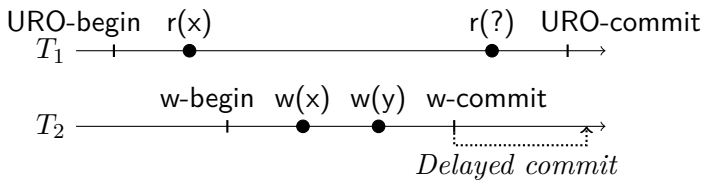

Figure 3.1: In order to preserve consistency, the write back of shared variables updated by an update transaction must be delayed until after any URO transaction has completed execution.

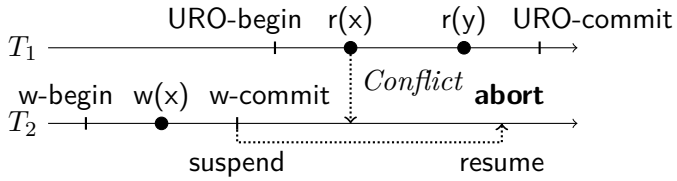

Figure 3.2: A read access to a shared variable updated by a suspended update transaction will abort the latter (when it resumes).

data (URO path). This provides the key benefit of ensuring strong progress guarantees for read-only transactions, which are spared by spurious (and repeated) aborts caused by the underlying HTM implementation.

Let us assume, for simplicity, that update transactions execute only using HTM (the case of ROT-based update transactions is analogous and will be discussed more in detail in Section 3.4.3). HTM transactions (and ROTs) buffer memory writes until the point of commit, hence, concurrent read-only transactions can safely execute with update transactions, as long as the latter ones do not commit. In fact, any read performed by a URO after a conflicting write of a concurrent update transaction will immediately abort the latter.

However, it is unsafe for update transactions to commit when there are concurrent UROs. This is illustrated in Figure 3.1 where an uninstrumented read-only transaction (T_1) and an update transaction (T_2) concurrently access two shared variables. As T_2 fully executes between two read accesses by T_1, T_2 cannot detect the concurrent execution of T_1 and, by committing, T_2 would expose T_1 to an inconsistent snapshot that may contain a mix of old and new values (if $r(?) \neq r(y)$ in the figure). To overcome this problem, a key idea in P8TM is to suspend the hardware speculation of an update transaction, and then wait for all current UROs to complete by using an RCU-like (epoch-based) quiescence mechanism [36–38]. This suspend-wait sequence has a two-fold effect. First, it drains all current read-only transactions that may read a location written by a suspended update transaction, as these may be exposed to inconsistent snapshots if the update transaction committed before their completion (as illustrated in Figure 3.1. Second, any read issued

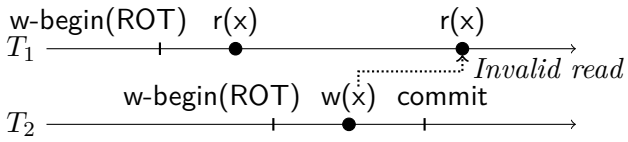

Figure 3.3: ROTs do not track reads and may, as such, observe different values when reading the same variable multiple times.

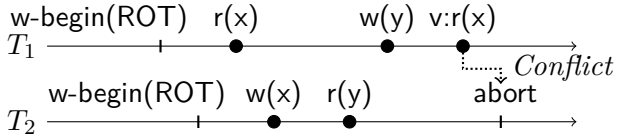

Figure 3.4: By re-reading x during *rot-rset* validation at commit time (denoted by v:r), T_1 forces the abort of T_2 that has updated x in the meantime.

to a location previously written by a suspend hardware transaction will cause the abort of the latter, as illustrated in Figure 3.2. As a result, after the wait is complete, it is safe to commit the update transaction, so P8TM simply resumes hardware speculation and issues a commit request.

3.3.2 Touch-based Validation

Touch-To-Validate (T2V) is another core mechanism of P8TM that enables safe and concurrent execution of ROT-based update transactions. As already mentioned, in fact, ROTs do not track read accesses in hardware. As such, their concurrent execution is generally unsafe, as illustrated by the example in Figure 3.3. Thread T_1 starts a ROT and reads x. At this time, thread T_2 starts a concurrent ROT, writes a new value to x, and commits. As ROTs do not track reads, they are unable to detect write-after-read conflicts. As such, the ROT of T_1 does not get aborted and can read inconsistent values (e.g., the new value of x). To avoid such scenarios T2V leverages two key mechanisms that couple: *(i)* software-based tracking of read accesses; and *(ii)* hardware- and software-based read-set validation during the commit phase.

For the sake of clarity, assume that threads only execute ROTs — other execution modes later will be considered later. A thread can be in one of three states: *inactive*, *active*, and *committing*. A thread that executes non-transactional code is inactive. When the thread starts a ROT, it enters the active phase and starts tracking, in software, each read access to shared variables by logging the associated memory address in a special data structure called *rot-rset*. Finally, when the thread finishes executing its transaction, it enters the committing phase. At this point, it has to wait for concurrent threads that are in the active phase to either enter the commit phase or become inactive (upon abort). Thereafter, the committing thread traverses its *rot-rset* and re-reads each address before eventually committing.

The goal of this validation step is to "touch" each previously read memory location in order to abort any concurrent ROT that might have written to the same address. For example, in Figure 3.4, T_1 re-reads x during *rot-rset* validation. At that time, T_2 has concurrently updated x but has not yet committed, and it will therefore abort (remember that ROTs track and detect conflicts for writes). This allows T_1 to proceed without breaking consistency: indeed, ROTs buffer their updates until commit and hence the new value of x written by T_2 is not visible to T_1. Note that adding a simple quiescence phase before commit, without performing the *rot-rset* validation, cannot solve the problem in this scenario.

The originality of the T2V mechanism is that the ROT does not use read-set validation for verifying that its read-set is consistent, as many STM algorithms do, but to trigger hardware conflicts detection mechanisms. This also means that the values read during *rot-rset* validation are irrelevant and ignored by the algorithm.

3.4 Algorithm

This section provides a detailed description of P8TM's algorithm. For the sake of clarity, P8TM is presented in an incremental fashion. First, describing the management of the URO path (Section 3.4.1) and of the ROT path (Section 3.4.2), each on its own. Then, in Section 3.4.3 provides a complete description of the algorithm, by discussing *(i)* how to extend the ROT path to first attempt using HTM transactions, and *(ii)* how to synchronize the URO and ROT paths with the pessimistic fallback path (based on a single global lock).

Finally, in Section 4.4.1, the correctness of the proposed solution is discussed.

3.4.1 URO Path

Let us start by considering an initial version of the P8TM algorithm (Algorithm 1) that assumes that read-only transactions execute in the URO path, and that update transactions execute using plain HTM transactions. For simplicity, this version of the algorithm blindly retries failed update transactions, irrespective of the abort cause.

To ensure proper synchronization with update transactions, P8TM must keep track of which UROs are executing. This is achieved by having every thread maintain a status variable, that is set and unset in the BEGIN_RO() and COMMIT_RO() functions when respectively starting and ending a read-only transaction.

Update transactions are started and committed by calling the BEGIN() and COMMIT() functions. They execute as plain HTM transactions, hence, throughout the execution of an update transaction, the memory writes are buffered and, thus, hidden from UROs.

Assume there is an update transaction and a concurrent URO that, respectively, update and read the same shared variable. If the memory access in the URO path occurs *after* the update transaction has written the variable, then the update transaction will immediately abort and restart. If however, the read occurs *before* the update transaction issues the write access, then no conflict will be detected and the URO will be serialized before the update transaction.

Algorithm 1 P8TM: URO path only algorithm

```
1: Shared variables:
2:     status[N] ← {⊥, ⊥, ..., ⊥}                    ▷ One per thread
3: Local variables:
4:     tid ∈ [0..N]                                   ▷ Identifier of current thread
5: function SYNCHRONIZE
6:     s[N] ← status                                  ▷ Read all statuses
7:     for i ← 0 to N−1 do                            ▷ Wait until all threads...
8:         if s[i] is ACTIVE then                     ▷ ...running UROs...
9:             wait until status[N] ≠ s[i]            ▷ ...end
10: end function
11: function BEGIN_RO
12:     status[tid] ← ACTIVE                           ▷ Update thread's status
13:     MEM_FENCE                                      ▷ Ensure visibility to update txs.
14: end function
15: function COMMIT_RO
16:     status[tid] ← ⊥                                ▷ Reset thread's status
17: end function
18: function BEGIN_W                                   ▷ Start update tx.
19:     repeat until TX_BEGIN = STARTED
20: end function
21: function COMMIT_W
22:     TX_SUSPEND                                     ▷ Suspend transaction
23:     SYNCHRONIZE                                    ▷ Let UROs drain their reads
24:     TX_RESUME                                      ▷ Resume transaction
25:     TX_COMMIT                                      ▷ Write back updates
26: end function
```

When an update transaction completes its execution, it must issue a commit request in order to write back its (speculative) updates. Yet, doing so without precaution would break consistency, since a URO might see a mix of old and new data (prior and after the commit of the update transaction).

Therefore, before commit, an update transaction waits for all UROs that *might* have read any of the locations it has written to. Since P8TM does not keep track of which memory locations have been accessed by UROs (which would require software instrumentation of memory accesses), it relies on a lightweight, RCU-like, quiescence mechanism that waits for the completion of any URO found active at the beginning of the quiescence phase. This is implemented in the SYNCHRONIZE() function by reading the status of each thread once and waiting for all *active* to change value. Note that this quiescence mechanism does not prevent the start of new read-only transactions, nor forces a suspended update transaction to wait for read-only transactions activated after the start of its quiescence phase. This is safe, since read-after-write conflicts will be handled as described above, i.e., by aborting the update transaction.

An additional challenge is that the quiescence barrier cannot be implemented straightforwardly in the context of hardware transaction. The problem is that if a URO updates its status that is being monitored by some concurrent update transaction, this will be detected as a read-write conflict, and lead to the abort of the update transaction.

To tackle this issue, P8TM exploits the S/R mechanism of the POWER8 processor,

which allows to temporarily suspend the active transaction, perform non-transactional operations, and later resume the transaction. P8TM relies on this feature to execute the quiescence phase and allow update transactions to monitor the status of concurrent UROs without incurring spurious aborts.

Note that any conflict occurring while a transaction is suspended will trigger an abort upon its resume, hence protecting concurrent UROs from seeing inconsistent snapshots. Indeed, consider a URO that starts after the call to SYNCHRONIZE(), i.e., which has not been found active by an update transaction upon the start of its quiescence phase. This URO will execute concurrently with the write-back phase of the update transaction. If the URO reads any memory location that has been updated by the update transaction before this completes its write-back phase (which is atomic), then the latter will abort; else, if the read is issued after the completion of the write-back phase, the URO will see the new version.

3.4.2 ROTs Path

We now present a version of the P8TM algorithm (Algorithm 2) assuming only the existence of update transactions running in the ROT path. Also in this case, for simplicity, ROTs blindly retry to execute failed attempts irrespective of the abort cause.

To start an update transaction, a thread first lets others know that it is *active* and initializes its data structures before actually starting a ROT (Lines 8–11). Then, during ROT execution, it just keeps track of reads to shared data by adding them to the thread-local *rot-rset* (Line 15). To complete the ROT, the thread first announces that it is *committing* by setting its shared *status* variable. Note that this is performed while the ROT is suspended (Lines 28–31) because otherwise the write would be buffered and invisible to other threads.

Next, the algorithm quiesces by waiting for all threads that are in a ROT to at least reach their commit phase (Lines 17–22). It then executes the touch-based validation mechanism, which simply consists in re-reading all address in the *rot-rset* (Lines 23–26), before finally committing the ROT (Line 34) and resetting the *status*.

3.4.3 Complete Algorithm

The naive approach of the basic algorithm to only use ROTs is unfortunately not practical nor efficient in real-world settings for two main reasons: (1) ROTs only provide "best effort" properties and thus a fallback is needed to guarantee liveness; and (2) using ROTs for short transactions that fit in a regular HTM transaction is inefficient because of the overhead of the software-based read tracking and validation mechanisms. Therefore, the algorithm is extended so that if first tries to use regular transactions, then upon failure switches to ROTs, and finally falls back to a global lock (GL) in order to guarantee progress. The pseudo-code of the complete algorithm is shown in Algorithms 3, 4 and 5.

For HTM transactions and ROTs to execute concurrently, the former must delay their commit until completion of all active ROTs. This is implemented using an RCU-like quies-

Algorithm 2 P8TM: ROT path only algorithm

```
1: Shared variables:
2:     status[N] ← {⊥, ⊥, ..., ⊥}                          ▷ One per thread
3: Local variables:
4:     tid ∈ [0..N]                                        ▷ Identifier of current thread
5:     rot-rset ← ∅                                        ▷ Transaction's read-set
6: function BEGIN_ROT
7:     repeat                                              ▷ Blindly retry ROT
8:         status[tid] ← ACTIVE                            ▷ Update status
9:         MEM_FENCE                                       ▷ Make sure others know
10:        rot-rset ← ∅                                    ▷ Clear read-set
11:        tx ← TX_BEGIN_ROT                               ▷ Start ROT
12:    until tx = STARTED                                  ▷ Repeat until success...
13: end function
14: function READ(addr)                                    ▷ Read shared variable
15:    rot-rset ← rot-rset ∪{addr}                         ▷ Track ROT reads
16: end function
17: function SYNCHRONIZE
18:    s[N] ← status                                       ▷ Read and copy all status variables
19:    for i ← 0 to N−1 do                                 ▷ Wait until all threads...
20:        if s[i] = ACTIVE then                           ▷ ...that are active...
21:            wait until status[i] ≠ s[i]                 ▷ ...end
22: end function
23: function TOUCH_VALIDATE
24:    for addr ∈ rot-rset do                              ▷ Re-read all elements...
25:        read addr                                       ▷ ...from read-set
26: end function
27: function COMMIT_ROT
28:    TX_SUSPEND                                          ▷ Suspend ROT
29:    status[tid] ← ROT-COMMITTING                        ▷ Tell others...
30:    MEM_FENCE                                           ▷ ...we are committing
31:    TX_RESUME                                           ▷ Resume ROT
32:    SYNCHRONIZE                                         ▷ Quiescence inside ROT
33:    TOUCH_VALIDATE                                      ▷ Touch to validate
34:    TX_COMMIT_ROT                                       ▷ Commit ROT
35:    status[tid] ← ⊥
36: end function
```

cence mechanism as in the URO algorithm (Lines 12–17). Note that a simple quiescence, without a validation step afterwards, is sufficient in this case.

In this version, aborted hardware transactions and ROTs are blindly retried. Conversely, the status code returned by TX_BEGIN/TX_BEGIN_ROT is exploited to determine which retry policy to use. If the return code of TX_BEGIN/TX_BEGIN_ROT is `STARTED`, indicating success, the HTM transaction/ROT can start executing speculatively. If an abort happens during execution of a HTM transaction/ROT, then controls jumps back to just after the call to TX_BEGIN/TX_BEGIN_ROT and the status code contains information about the failure cause. For the sake of simplicity, assume that the status code can be `STARTED`, `TRANSIENT-ABORT`, or `CAPACITY-ABORT` to respectively indicate if the transaction executes speculatively, or has aborted due a problem that is unlikely (e.g., contention) or likely (capacity) to be encountered again in a subsequent attempt.

Algorithm 3 P8TM: complete algorithm

```
1: Shared variables:
2:     status[N] ← {⊥, ⊥, ..., ⊥}                          ▷ One per thread
3:     glock ← FREE                                        ▷ Spin lock to serialize transactions
4: Local variables:
5:     tid ∈ [0..N]                                        ▷ Identifier of current thread
6:     mode ∈ {HTM, ROT, GL}                               ▷ Transaction mode
7:     rot-rset ← ∅                                        ▷ Transaction's read-set
8: function READ(addr)                                     ▷ Read shared variable
9:     if mode = ROT then
10:        rot-rset ← rot-rset ∪{addr}                     ▷ Track ROT reads
11: end function
12: function SYNCHRONIZE
13:    s[N] ← status                                       ▷ Read and copy all status variables
14:    for i ← 0 to N−1 do                                 ▷ Wait until all threads...
15:        if s[i] = ACTIVE                                ▷ ...that are active...
       ..... ∨ (mode = GL ∧ s[i] = ROT-COMMITTING) then
16:            wait until status[i] ≠ s[i]                 ▷ ...cross barrier
17: end function
18: function TOUCH_VALIDATE
19:    for addr ∈ rot-rset do                              ▷ Re-read all elements...
20:        read addr                                       ▷ ...from read-set
21: end function
22: function BEGIN_W
23:    wait until glock = FREE                             ▷ Global lock must be free
24:    BEGIN_HTM                                           ▷ Try HTM first
25:    if mode ≠ HTM then                                  ▷ If HTM fails...
26:        BEGIN_ROT                                       ▷ ...fall back to ROT
27:        if mode ≠ ROT then                              ▷ If ROT also fails...
28:            BEGIN_GL                                    ▷ ...default to global lock
29: end function
30: function BEGIN_HTM
31:    trials ← 0
32:    repeat                                              ▷ Retry HTM a few times
33:        trials ← trials+1
34:        tx ← TX_BEGIN                                   ▷ HTM begin
35:        if tx = STARTED then                            ▷ Success?
36:            if glock ≠ FREE then                        ▷ Add lock to read-set
37:                TX_ABORT                                ▷ Abort if lock taken
38:            mode ← HTM                                  ▷ Run in HTM mode
39:    until mode = HTM                                    ▷ Repeat until success...
       ..... ∨ tx = CAPACITY-ABORT                         ▷ ...or capacity abort...
       ..... ∨ trials > MAX-HTM-TRIALS                     ▷ ...or too many trial
40: end function
```

Algorithm 4 P8TM: complete algorithm (2)

```
1: function BEGIN_ROT
2:     trials ← 0
3:     repeat                                                  ▷ Retry ROT a few times
4:         trials ← trials+1
5:         status[tid] ← ACTIVE                                ▷ Update status
6:         MEM_FENCE                                           ▷ Make sure others know
7:         if glock ≠ FREE then                                ▷ Global lock taken?
8:             status[tid] ← ⊥                                 ▷ Yes: defer to GL...
9:             wait until glock = FREE                         ▷ ...wait...
10:            go to 5                                         ▷ ...and retry
11:        rot-rset ← ∅                                        ▷ Clear read-set
12:        tx ← TX_BEGIN_ROT                                   ▷ HTM ROT begin
13:        if tx = STARTED then                                ▷ Success?
14:            mode ← ROT                                      ▷ Run in ROT mode
15:    until mode = ROT                                        ▷ Repeat until success...
       ..... ∨ tx = CAPACITY-ABORT                             ▷ ...or capacity abort...
       ..... ∨ trials > MAX-HTM-TRIALS                         ▷ ...or too many trial
16: end function
17: function BEGIN_GL
18:    status[tid] ← ⊥                                         ▷ Not using TM
19:    MEM_FENCE                                               ▷ Make sure others know
20:    repeat                                                  ▷ Acquire global lock
21:        wait until glock = FREE                             ▷ Test and...
22:    until CAS(glock, FREE, LOCKED)                          ▷ ...test and set
23:    mode ← GL                                               ▷ Run in GL mode
24:    SYNCHRONIZE                                             ▷ Perform quiescence phase
25: end function
26: function COMMIT_W
27:    switch mode do
28:        case HTM
29:            TX_SUSPEND                                      ▷ Suspend transaction
30:            SYNCHRONIZE                                     ▷ Perform quiescence phase
31:            TX_RESUME                                       ▷ Resume transaction
32:            TX_COMMIT                                       ▷ End transaction
33:        case ROT
34:            TX_SUSPEND                                      ▷ Suspend transaction
35:            status[tid] ← ROT-COMMITTING                    ▷ Tell others...
36:            MEM_FENCE                                       ▷ ...we are committing
37:            TX_RESUME                                       ▷ Resume transaction
38:            SYNCHRONIZE                                     ▷ Quiescence inside ROT
39:            TOUCH_VALIDATE                                  ▷ Touch to validate
40:            TX_COMMIT                                       ▷ End transaction
41:            status[tid] ← ⊥
42:        case GL
43:            glock ← FREE                                    ▷ Release global lock
44: end function
```

Algorithm 5 P8TM: URO path

```
1: function BEGIN_RO
2:     status[tid] ← ACTIVE                          ▷ Update thread's status
3:     MEM_FENCE                                     ▷ Ensure visibility to update txs.
4:     if glock ≠ FREE then                          ▷ Global lock taken?
5:         status[tid] ← ⊥                           ▷ Yes: defer to GL...
6:         wait until glock = FREE                   ▷ ...wait...
7:         go to 2                                   ▷ ...and retry
8: end function
9: function COMMIT_RO
10:    status[tid] ← ⊥                               ▷ Reset thread's status
11: end function
```

Transactions try to run in HTM and ROT modes a limited number of times, switching immediately if the cause of the failure is a capacity abort (Lines 39 and 15). The GL fallback uses a basic spin lock, which is acquired upon transaction begin (Lines 20–21) and released upon commit (Line 43). Observe that the quiescence mechanism must also be called after acquiring the lock to wait for completion of ROTs that are in progress and might otherwise see inconsistent updates (Line 24), and that the GL path must actually wait for ROTs to fully complete, not just enter the commit phase as for the other execution modes (Line 15). The rest of the algorithm is relatively straightforward.

To understand the intuition behind how URO path can be integrated safely with concurrent update transactions, consider first that transactions that do not modify shared data cannot cause the abort of a HTM transaction or a ROT. Furthermore, because HTM transactions and ROTs buffer their writes and quiesce before committing, they cannot propagate inconsistent updates to UROs.

Finally, GL and UROs cannot conflict with each other as long as they do not run concurrently. This is ensured by the quiescence phase after acquiring the global lock, and the fact that UROs do not start executing until the lock is free (Line 6). Note that, if the lock is taken, UROs defer to the update transaction holding the global lock by resetting their status (Line 5) before waiting for the lock to be free and retrying the whole procedure. Otherwise P8TM could run into a deadlock situation with a URO waiting for the lock held by a GL transaction, while the latter is blocked in quiescence waiting for the former to complete its execution.

3.4.4 Correctness Argument.

When the GL path is active, concurrency is disabled. This is guaranteed since: *(i)* transactions in HTM path subscribe eagerly to the GL, and are thus aborted upon the activation of this path; *(ii)* after the GL is acquired, a quiescence phase is performed to wait for active ROTs or UROs. Atomicity of a transaction in the HTM path is provided by the hardware against concurrent HTM transactions/ROTs and by GL subscription.

As for the UROs, the quiescence mechanism guarantees two properties:

- UROs activated after the start of an update transaction T, and before the start of T's

quiescence phase, can be safely serialized before T because they are guaranteed not to see any of T's updates, which are only made atomically visible when the corresponding HTM transaction/ROT commits;

- UROs activated after the start of the quiescence phase of an update transaction T can be safely serialized either after T because they are guaranteed to abort T in case they read a value written by T before T commits, or after T as they will see all the updates produced by T's commit. It is worth noting here though that this is only relevant when a URO may conflict with T, in case of disjoint operation both serialization orders are equivalent.

Now we are only left with transactions running on the ROT path. The same properties of quiescence for UROs apply here and avoid ROTs reading inconsistent states produced by concurrent HTM transactions. Nevertheless, since ROTs do modify the shared state, they can still produce non-serializable histories; such as the scenario in Figure 2. Assume a ROT, say T_1, issued a read on X, developing a read-write conflict, with some concurrently active ROT, say T_2. There are two cases to consider: T_1 commits before T_2, or vice-versa.

If T_1 commits first, then if it reads X after T_2 (which is still active) wrote to it, then T_2 is aborted by the hardware conflict detection mechanism. Else, we are in presence of a write-after-read conflict. T_1 finds $status[T_2] := ACTIVE$ (because T_2 issues a fence before starting) and waits for T_2 to enter its commit phase (or abort). Then T_1 executes its T2V, during which, by re-reading X, would cause T_2 to abort.

Consider now the case in which T_2 commits before T_1. If T_1 reads X, as well as any other memory position updated by T_2, before T_2 writes to it, then T_1 can be safely serialized before T_2 (as T_1 observed none of T_2's updates). If T_1 reads X, or any other memory position updated by T_2, after T_2 writes to it and before T_2 commits, then T_2 is aborted by the hardware conflict detection mechanism; a contradiction. Finally, it is impossible for T_1 to read X after T_2 commits: in fact, during T_2's commit phase, T_2 must wait for T_1 to complete its execution; hence, T_1 must read X after T_2 writes to it and before T_2 commits, falling in the above case and yielding another contradiction.

3.5 Read-set Tracking

The T2V mechanism requires to track the read-sets of ROTs for later replaying them at commit time. The implementation of the read-set tracking scheme is crucial for the performance of P8TM. In fact, as discussed in Section 2.2.5, ROTs do not track loads at the TMCAM level, but they do track stores and the read-set tracking mechanism must issue stores in order to log the addresses read by a ROT. The challenge, hence, lies in designing a software mechanism that can exploit the TMCAM's capacity in a more efficient way than the hardware would do. Two alternative mechanisms that tackle this challenge by exploring different trade-offs between computational and space efficiency are described next.

Time-efficient implementation uses a thread local, cache aligned array, where each entry is used to track a 64-bit address. Since the cache lines of the POWER8 CPU are 128 bytes long, this means that 16 consecutive entries of the array, each storing an arbitrary address, will be mapped to the same cache line and occupy a single TMCAM entry. Therefore, this approach allows for fitting up to 16× larger read-sets within the TMCAM as compared to the case of HTM transactions. Given that they track 64 cache lines, thread-local arrays are statically sized to store exactly 1024 addresses. It is worth noting here that since conflicts are detected at the cache line level granularity, it is not necessary to store the 7 least significant bits, as addresses point to the same cache line. However, this optimization is omitted as this will add extra computational overhead, yielding a space saving of less than 10%.

Space-efficient implementation seeks to exploit the spatial data locality in the application's memory access patterns to compress the amount of information stored by the read-set tracking mechanism. This is achieved by detecting a common prefix between the previously tracked address and the current one, and by storing only the differing suffix and the size (in bytes) of the common prefix. The latter can be conveniently stored using the 7 least significant bits of the suffix, which, as discussed, are unnecessary. With applications that exhibit high spatial locality (e.g., that sequentially scan memory), this approach can achieve significant compression factors with respect to the time-efficient implementation. However, it introduces additional computational costs, both during the logging phase (to identify the common prefix) and in the replay phase (as addresses need to be reconstructed).

3.6 Self-tuning

In workloads where transactions fit the HTM's capacity limitations, P8TM still forces HTM transactions to incur the overhead of S/R, in order to synchronize them with possible concurrent ROTs. In these workloads, the ideal decision would be to just disable the ROT path, so to spare the HTM path from any overhead. However, it is not trivial to determine when it is beneficial to do so; this is workload dependent and it can be hard to determine via static analysis, especially in applications that make intensive use of pointers.

This issue is addressed by integrating into P8TM a self-tuning mechanism based on a lightweight reinforcement learning technique, UCB [110], which will be described shortly. UCB determines, in an automatic fashion, which of the following modes to use: (M1) HTM falling back to ROT, and then to GL; (M2) HTM falling back directly to the GL; (M3) starting directly in ROT before falling back to the GL. Note that UROs and ROTs impose analogous overheads to HTM transactions. Thus, in order to reduce the search space to be explored by the self-tuning mechanism, whenever the ROT path is disabled, the URO path is also disabled. In such cases read-only transactions are treated as update transactions.

Figure 3.5 shows the three paths and the rules for switching between them. When ROTs are disabled (M1⇒M2), the quiescence call within transactions can be skipped only once there are no more active ROTs. When switching from M2 to M3, instead, it is enough

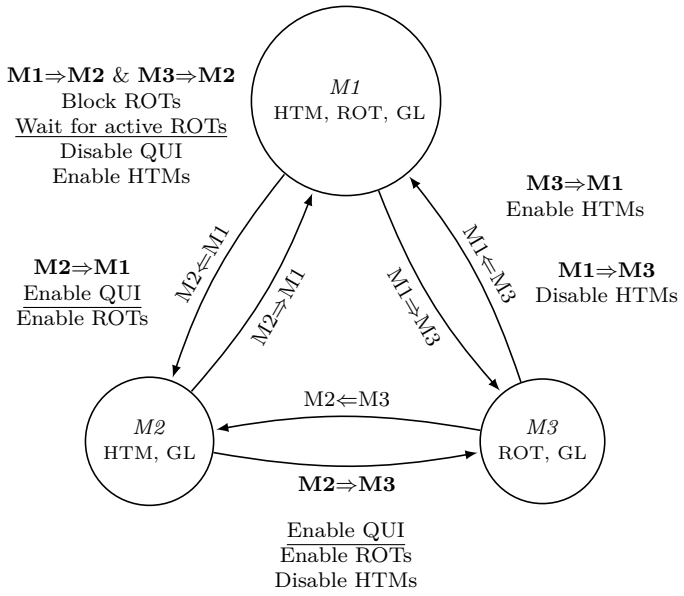

Figure 3.5: Different execution paths that can be used by transactions and rules for switching between them. Lines within rules represent necessary memory barriers.

to enable ROTs after having ensured, via a memory fence, that all threads are informed about the need to enable quiescence. This forces any active HTM transaction to perform the quiescence once it reaches its commit phase, while it will abort any active transaction that has reached commit stage but has not yet committed (since the flag is already part of the read-set). The other rules are straightforward.

UCB

Upper confidence bounds (UCB) [90] is a provably optimal solution to the multiarmed bandit problem [111], i.e., a classical reinforcement-learning problem that embodies the trade-off between exploration and exploitation. In the multiarmed bandit, a gambler tries to maximize the reward obtained from playing different levers of a multiarmed slot machine, where the levers' rewards are random variables with a priori unknown distributions. The use of the UCB technique is motivated by the strong theoretical guarantees[1] while imposing negligible computational overheads. After an initial phase, in which every lever is sampled once, UCB estimates the expected reward for lever i as $\bar{x}_i + \sqrt{(2\log n)/n_i}$, where: $\bar{x}_i$ is the average reward for lever i; n is the number of the current trial; and n_i is the number of times the lever i has been tried.

In order to use UCB in P8TM, each execution mode is associated with a different lever, and its reward to the throughput obtained by using that mode during a sampling interval

[1]Logarithmic bounds on the cumulative error, called regret, from playing non-optimal levers even in finite time horizons [90].

of 100 microseconds.

3.7 Evaluation

This section shows the evaluation of P8TM against state of the art TM systems using a set of synthetic micro-benchmarks and complex, real-life applications. It first starts by evaluating both variants of read-set tracking to show how they are affected by the size of transactions and degree of contention. A sensitivity analysis aimed to investigate various factors that affect the performance of P8TM is then is conducted. To this end, a micro-benchmark that manipulates a hashmap via lookup, insert, and delete transactions was used. Finally, P8TM is tested using two complex, realistic workloads: the popular STAMP benchmark suite and a port to the TM domain of the TPC-C benchmark for in-memory databases.

The following baselines were considered: *(i)* plain HTM with a global lock fallback (HTM-SGL), *(ii)* NOrec with write back configuration, and finally *(iii)* the Hybrid NOrec algorithm with three variables to synchronize transactions and NOrec fallback (HyNOrec).

Regarding the retry policy, HTM path is executed 10 times and the ROT path is executed 5 times before falling back to the next path, except upon a capacity abort when the next path is directly activated. These values and strategies were chosen after doing an extensive offline experiment and selecting the best on average with different number of retries and different capacity aborts handling policies (e.g., fallback immediately vs treating it as a normal abort). All results presented in this section represent the mean value of at least 5 runs. The experiments were conducted on a machine equipped with IBM POWER8 8284-22A processor that has 10 physical cores, with 8 hardware threads each. The source code was compiled with GCC 6.2.1 using `-O2` flag on top of Fedora 24 with Linux 4.5.5. Thread pinning was use to pin a thread per core at the beginning of each run for all the solutions, and threads were distributed evenly across the cores.

3.7.1 Read-set Tracking

The goal of this section is to understand the trade-off between the time-efficient and the space-efficient implementations of read-set tracking that were explained earlier in Section 3.5. Three variants of P8TM are compared: *(i)* time-efficient read-set tracking (TE), *(ii)* a variant of space-efficient read-set tracking that only checks for prefixes of length 4 bytes, and otherwise stores the whole address (SE), and finally *(iii)* a more aggressive version of space-efficient read-set tracking that looks for prefixes of either 6 or 4 bytes (SE++).

Throughout this section, the number of threads is fixed to 10 (number of physical cores) and the percentage of update transactions at 100%, disabled the self-tuning module, and varied the transaction length across orders of magnitude to stress the ROT-path.

First, an almost contention-free workload to highlight the effect of capacity aborts alone is considered. The speedup with respect to HTM-SGL, breakdown of commits and aborts for this workload is shown in the first row of Figure 3.6. The three variants of P8TM

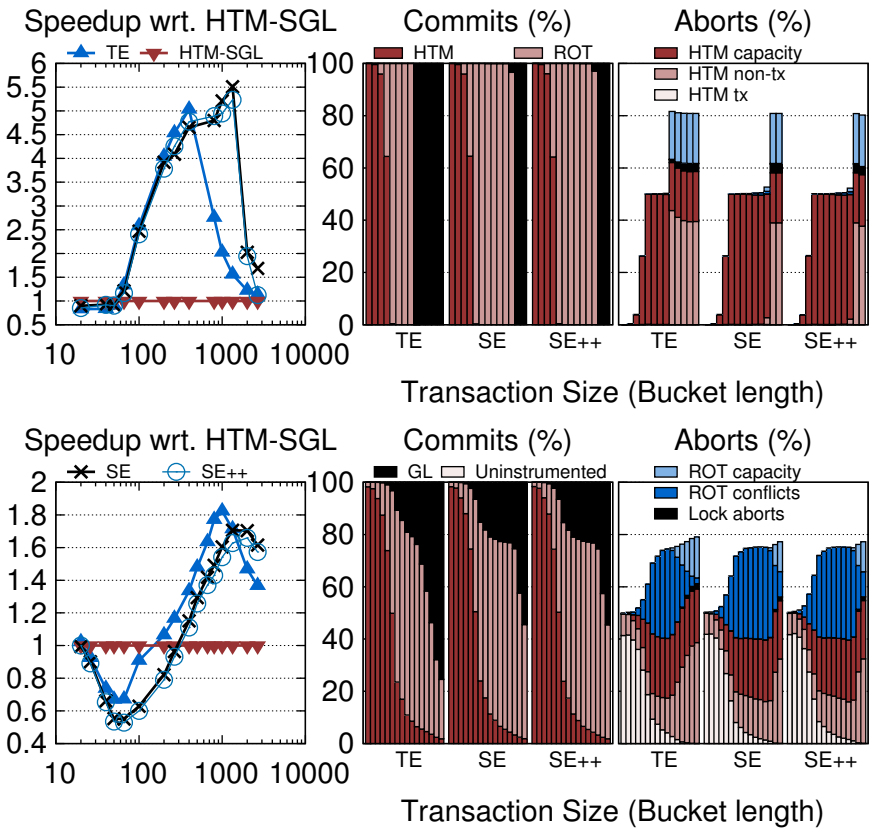

Figure 3.6: Evaluation of different implementations of read-set tracking.

achieve almost the same performance as HTM-SGL with small transaction sizes that fit inside regular HTM transactions, as seen from the commits breakdown. However, when moving to larger transactions, the three variants start outperforming HTM-SGL by up to 5.5× due to their ability to fit transactions within ROTs. The aborts breakdown in this region shows that all P8TM variants suffer from almost 50% capacity aborts when first executing in HTM, and almost no capacity aborts when using the ROT path. This shows the clear advantage of the T2V mechanism and how it can fit more than 10× larger transactions in hardware.

Comparing TE with SE and SE++, both space-efficient variants are able to execute even larger transactions as ROTs. Nevertheless, they incur an extra overhead, which is reflected as a slightly lower speedup than TE, before this starts to experience ROT capacity aborts; only then their ability to further compress the read-set within TMCAM pays off. Again, by looking at the commits and aborts breakdown, both space-efficient variants manage to commit all transactions as ROTs when TE is already forced to execute using the GL. Finally, when comparing SE and SE++, it can be concluded that trying harder to find longer prefixes is not useful, as in this workload there is a very low probability that the

accessed addresses share 6 bytes long prefixes.

The second row in Figure 3.6 shows the results for a workload that exhibits a higher degree of contention. In this case, with transactions that fit inside regular HTM transactions, HTM-SGL can outperform both SE and SE++ by up to 2× and TE by up to ∼30%. Since P8TM tries to execute transactions as ROTs after failing 10 times with HTM due to conflicts, the ROT path may be activated even in absence of capacity aborts; hence, the overhead of synchronizing ROTs and HTM transaction becomes relevant even with small transactions. With larger transactions, the computational costs of SE and SE++ are more noticeable in this workload where they are always outperformed by TE, as long as this is able to fit at least 50% of transactions inside ROTs. Furthermore, the gains of SE and SE++ with respect to TE are much lower when compared to the contention-free workload. From this, it can be deduced that TE is more robust to contention. This was also confirmed with the other workloads that will be discussed next.

3.7.2 Sensitivity Analysis

The results of a sensitivity analysis aimed to assess the impact of the following factors on P8TM's performance: *(i)* the size of transactions, *(ii)* the degree of contention, and *(iii)* the percentage of read-only transactions. Three dimensions were explored using the following configurations: *(i)* high capacity, low contention, *(ii)* high capacity, high contention, and *(iii)* low capacity, low contention, with 10%, 50%, and 90% update transactions.

The results for low capacity, high contention workload were skipped since they do not convey any extra information with respect to the low capacity, low contention scenario (which is actually even more favorable for HTM). These experiments compare three variants of P8TM to highlight the breakdown of gains from the various components: *(i)* P8TM$_{uro}$ executes read-only transactions as UROs while update transactions execute in HTM and fallback to a sequential ROT before acquiring the global lock, this highlights gains achievable through URO only, while, *(ii)* P8TM enables both UROs and concurrent ROTs with the TE read-set tracking, thus benefits from both UROs and T2V, and finally, *(iii)* P8TM$_{ucb}$ augments P8TM with the self-tuning to decide upon the most efficient execution path.

High capacity, low contention. Figure 3.7 shows the throughput, commits and abort breakdown for the high capacity, low contention configuration. For the read dominated workload, all variants of P8TM are able to outperform all the other TM solutions by up to 7×. This can be easily explained by looking at the commits breakdown, where variants of P8TM commit 90% of their transactions as UROs while the other 10% are committed as ROTs. However, since P8TM and P8TM$_{ucb}$ are capable of parallelizing ROTs, they achieve ∼2.4× higher throughput than P8TM$_{uro}$.

On the contrary, HTM-SGL commits only 10% of the transactions in hardware and falls back to GL in the rest, due to the high capacity aborts it incurs. It is worth noting that the decrease in the percentage of capacity aborts, along with the increase of number of threads, is due to the activation of the fallback path, which forces other concurrent transactions to abort.

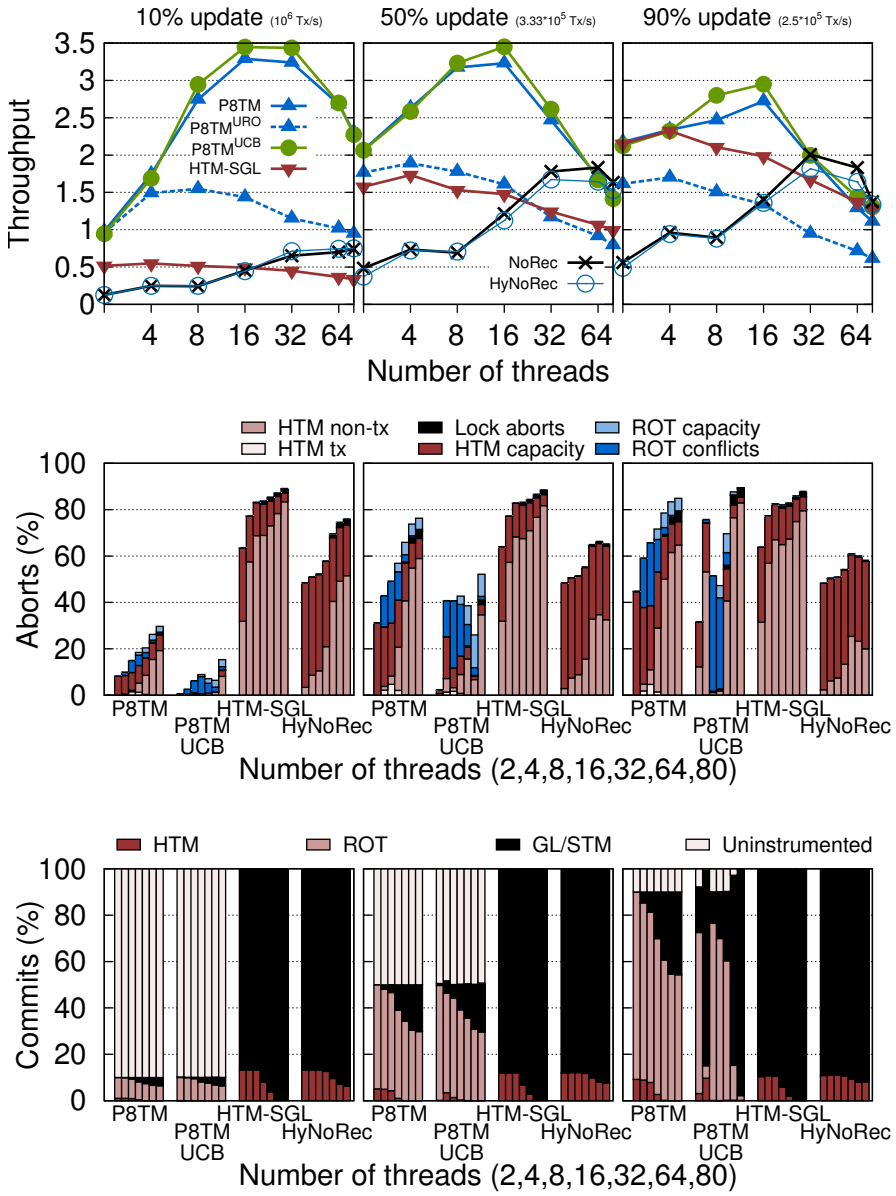

Figure 3.7: High capacity-low contention configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios.

Another interesting point is that P8TM$_{ucb}$ can outperform P8TM thanks to its ability to decrease the abort rate, as shown in the aborts breakdown. This is achieved by deactivating the HTM path, which spares P8TM from the cost of trying once in HTM before falling back to ROT (upon a capacity abort).

Workloads with more update transactions exhibit similar trends: P8TM and P8TM$_{ucb}$

outperform HTM-SGL by ∼2.2× and ∼1.4× in the 50% and 90% workloads, respectively. They also achieve the highest throughput in all workloads among all the considered baselines. The breakdown of commits plot demonstrates P8TM's ability to execute almost all update transactions using either HTMs or ROTs up to 8 threads, unlike HTM-SGL that only executes 10% of transactions in hardware. At high thread count, both NOrec and HyNOrec start to outperform both P8TM and P8TM$_{ucb}$, especially in the 90% workload. This can be explained by two reasons: *(i)* with larger numbers of threads there is higher contention on hardware resources (note that starting from 32 threads ROT capacity aborts start to become frequent) and *(ii)* the cost of quiescence becomes more significant as threads have to wait longer.

Despite that, P8TM variants achieve 2× and ∼1.4× higher throughput than NOrec and HyNOrec, when comparing their maximum throughputs regardless of the thread count.

High capacity, high contention. Figure 3.8 reports the results for the high capacity, high contention configuration. Trends for read dominated workloads are similar to the case of lower contention degree. However, scalability is much lower here due to the higher conflict rate. When considering the workloads with 50% and 90% update transactions, P8TM still achieves the highest throughput. Moreover, unlike in the low contention scenario, P8TM outperforms NOrec nor HyNOrec even at high thread count. This is due to the fact that handling contention is more efficiently done at the hardware level than in software. Although P8TM$_{uro}$ uses URO path to execute read-only transactions, it was unable to scale even in the 90% read-only workload, where its throughput is 2.5× lower than P8TM's. Again this is due to its inability to execute concurrent ROTs. This clearly indicates that T2V is beneficial even in read-dominated workloads.

Low capacity, low-contention. In workloads where transactions fit in HTM, it is expected that HTM-SGL will outperform all other TM solutions and that the overheads of P8TM will prevail. Results in Figure 3.9 confirm this expectation: HTM-SGL outperforms all other solutions, regardless of the ratio of read-only transactions, achieving up to 2.5× higher throughput than P8TM. However, P8TM$_{ucb}$, thanks to its self-tuning ability, is the, overall, best performing solution, achieving performance comparable to HTM-SGL at low thread count, and outperforming all other approaches at high thread count. The commits breakdown plots show that P8TM$_{ucb}$ does not commit any transaction using ROTs up to 8 threads, avoiding the synchronization overheads that, instead, affect P8TM.

It is worth noting, though, that P8TM, with 90% read-only transactions, does outperform HTM-SGL beyond 16 threads. By inspecting the breakdown of aborts and commits, it can be noticed that when hardware multithreading is enabled the performance of HTM-SGL deteriorates dramatically, due to the increased contention on hardware resources. Conversely, P8TM can still execute transactions in ROTs, hence achieving higher throughput.

Even though HyNOrec commits the same or higher percentage of HTM transactions than HTM-SGL, it is consistently outperformed by P8TM. This can be explained by look-

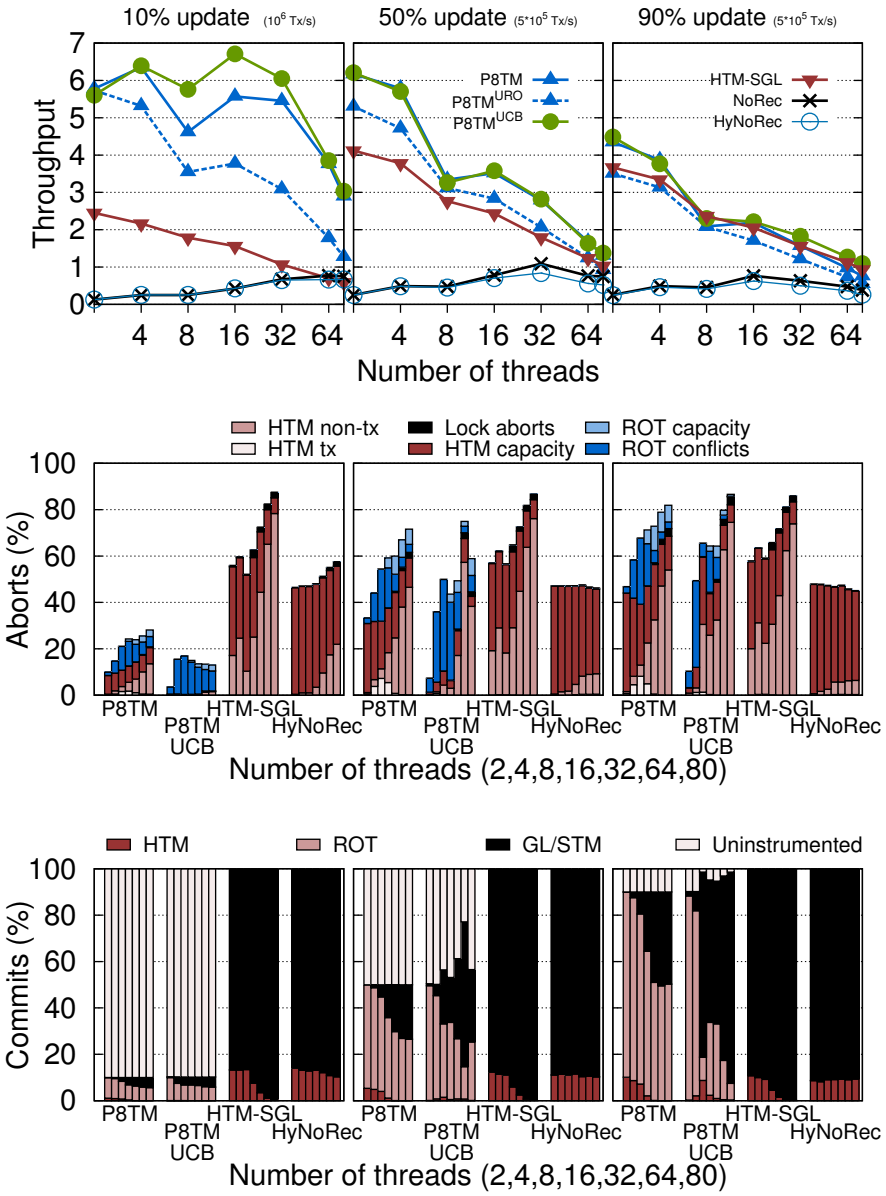

Figure 3.8: High capacity, high contention configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios.

ing at the performance of NOrec, which fails to scale due to the high instrumentation overheads it incurs with such short transactions. As for HyNOrec, its poor performance is a consequence of the inefficiency inherited by its NOrec fallback.

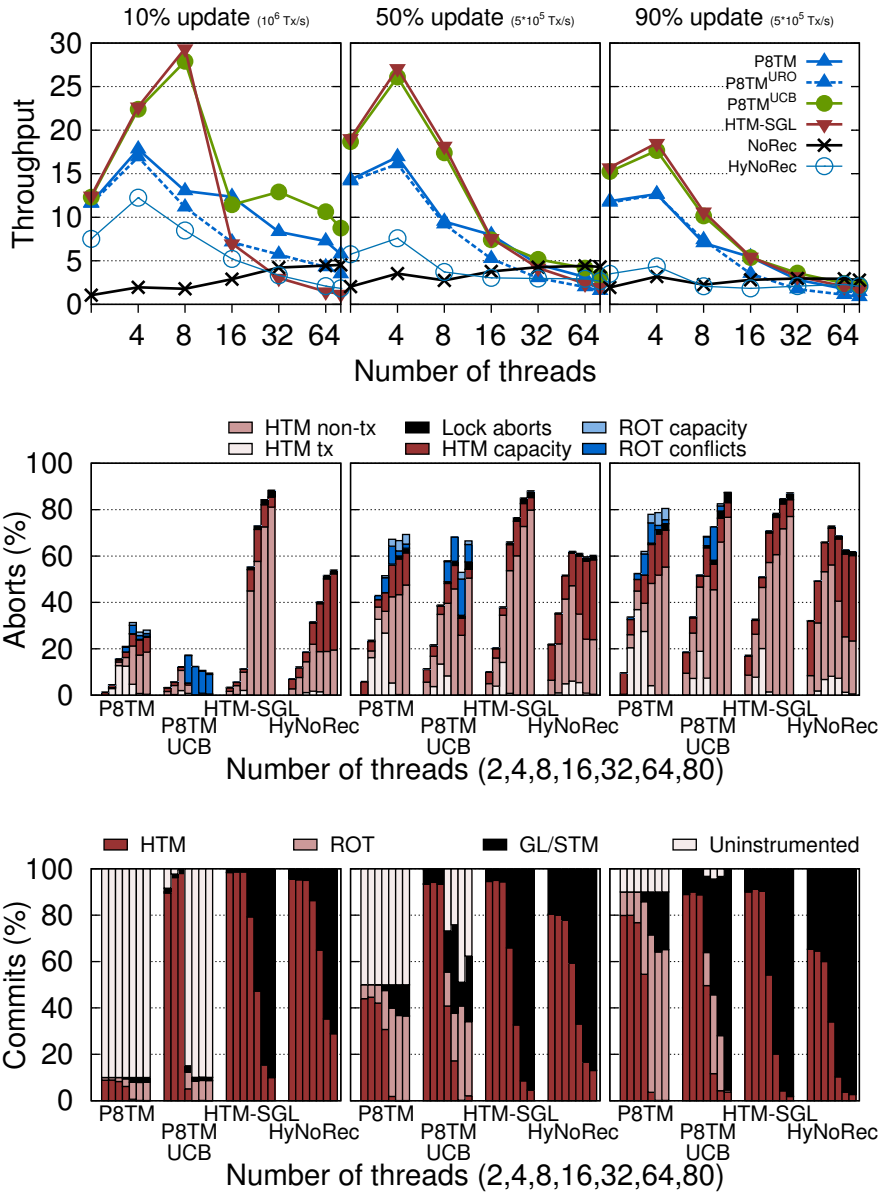

Figure 3.9: Low capacity, low contention configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios.

3.7.3 STAMP Benchmark Suite

All the applications of the STAMP suite (see Section 2.4) share a common trait: they do not have any read-only transactions. Therefore, P8TM will not utilize the URO path and any gain it can achieve stems solely from executing ROTs in parallel. The results of the

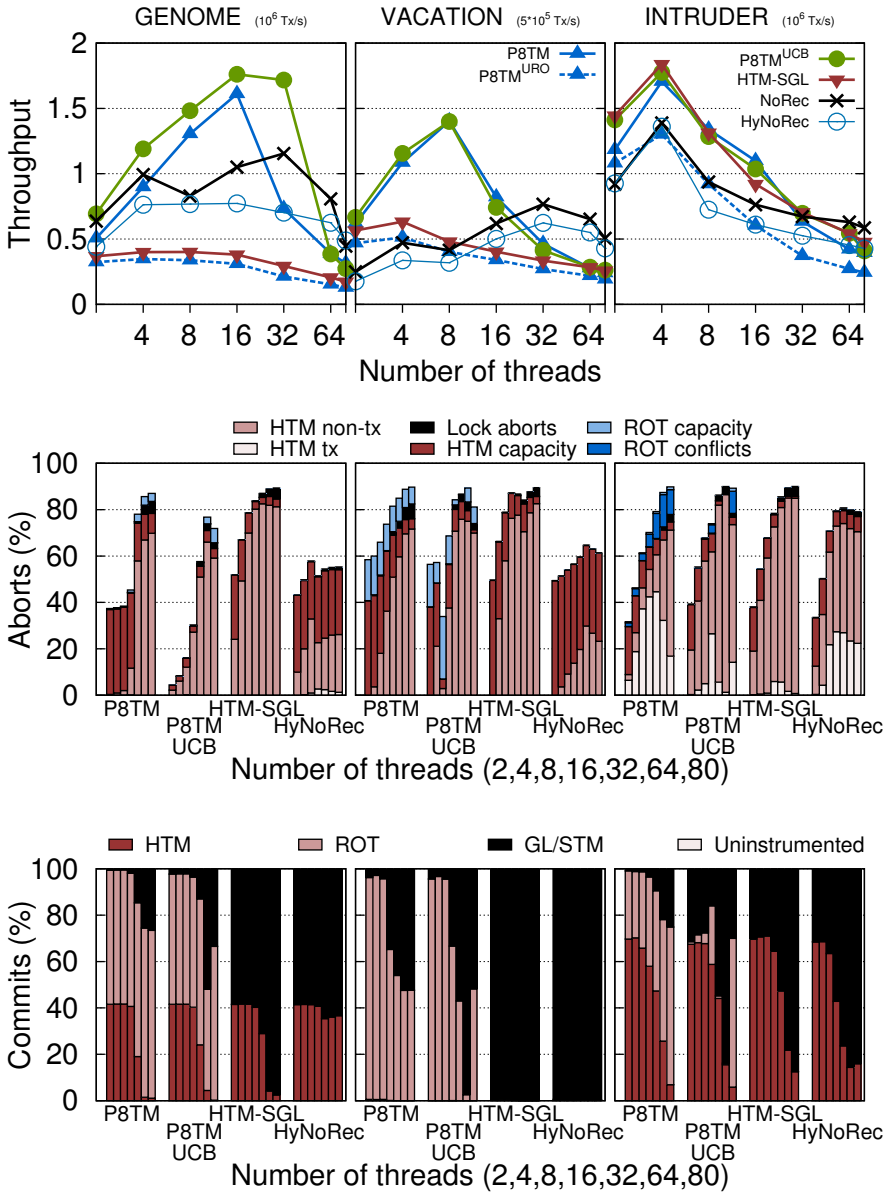

Figure 3.10: Throughput, abort rate, and breakdown of commit modes of STAMP benchmarks (1).

bayes application are omitted due to its high variance [112]. The *labyrinth* application is also omitted, as its transactions do not fit in neither HTM nor ROT—hence, exhibiting very similar performance trend to *yada*.

Both, **genome** and **vacation** are two applications with medium sized transactions and low contention; hence, they behave similarly to the previously analyzed high capacity, low

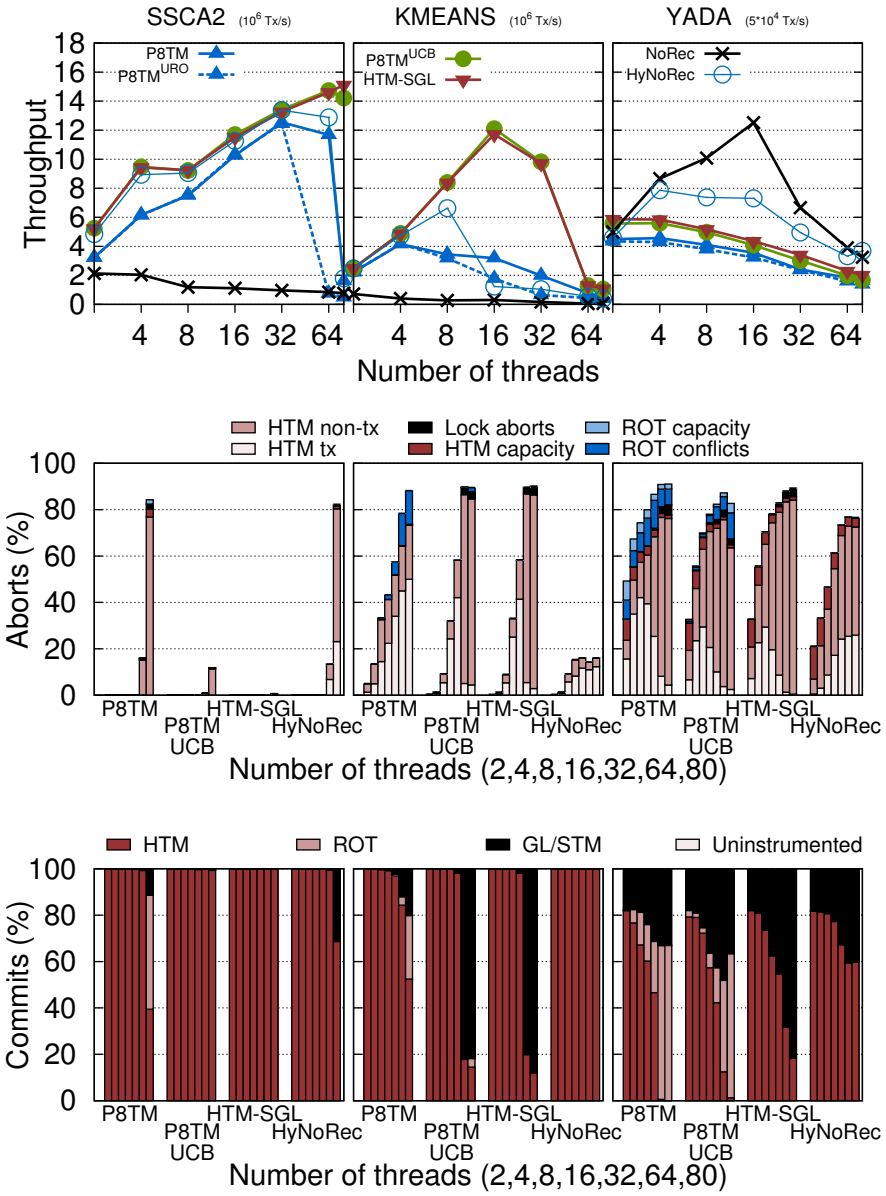

Figure 3.11: Throughput, abort rate, and breakdown of commit modes of STAMP benchmarks (2).

contention workloads. When looking at Figure 3.10, very similar to the workloads with high update ratios in Figure 3.7 can be seen. P8TM is capable of achieving the highest throughput and outperforming HTM-SGL by up to 4.5× in case of *genome* and ∼3.2× in the case of *vacation*. Again, P8TM$_{ucb}$ is even able to achieve higher throughput than P8TM due to deactivating the HTM path when capacity aborts are encountered, thus decreasing

the abort rate. When looking at the breakdown of commits, the ability of P8TM to execute most of transactions in either HTM or ROT at low thread counts is obvious. One difference between *genome* and *vacation* is that, in *vacation*, HTM-SGL never manages to commit transactions in hardware.

The same drawback is also noticed at high number of threads when comparing P8TM to NOrec and HyNOrec. Nevertheless, it is worth noting that the maximum throughput achieved by P8TM (at 16 threads) is 1.5× and 2× higher than NOrec (at 32 threads) in *genome* and *vacation*, respectively. This is due to instrumentation overheads of these solutions. These overheads are completely eliminated in case of write accesses within P8TM and are much lower for read accesses.

The **intruder** application generates transactions with medium read/write sets and high contention. This results in a similar performance for both P8TM and HTM-SGL: they achieve almost the same peak throughput at 8 threads and follow the same pattern with increasing number of threads. Although P8TM manages to execute all transactions as either HTM transactions or ROTs at low numbers of threads, given the low level of parallelism, the synchronization overheads incurred by P8TM are not outweighed by its ability to run ROTs concurrently. Nevertheless, P8TM$_{ucb}$ manages to overcome this limitation by disabling the ROT path and avoid these overheads. Both NOrec and HyNOrec were outperformed, which is again simply due to their high instrumentation costs.

The **ssca2** and **kmeans** applications generate transactions with small read/write sets and low contention. These are HTM friendly characteristics, and by looking at the throughput results in Figure 3.11, it can be seen that HTM-SGL is able to outperform all the other baselines and scale up to 80 threads in case of *ssca2* and up to 16 threads in case of *kmeans*. Although HyNOrec was able to achieve performance similar to HTM up to 32 threads in *ssca2* and 8 threads in *kmeans*, it was then outperformed due to the extra overheads it incurs to synchronize with the NOrec fallback. These overheads lead to increased capacity aborts as seen in the aborts breakdown.

Although P8TM commits almost all transactions using HTM up to 64 threads, it performed worse than both HTM-SGL and HyNOrec in *ssca2* due to the costs of synchronization. An interesting observation is that the overhead is almost constant up to 32 threads, since up to 32 threads there are no ROTs running and the overhead of the quiescence call is dominated by the cost of suspending and resuming the transaction. At 64 and 80 threads P8TM started to suffer also from capacity aborts similarly to HyNOrec. This led to a degradation of performance, with HTM-SGL achieving 7× higher throughput at 80 threads. Similar trends can be seen for *kmeans*, however with different threads counts and with lower adverse effects for P8TM. Again, these are workloads where P8TM$_{ucb}$ comes in handy as it manages to disable the ROT path and thus tends to employ HTM-SGL, which is the most suitable solution for these workloads.

The **yada** application has long transactions, large read/write set and medium contention. This is an example of a workload that is not hardware friendly and where hardware solutions are expected to be outperformed by software based ones. Figure 3.11 shows the clear advantage of NOrec over any other solution, achieving up to 3× higher throughput than hardware based solutions. When looking at the commits and abort break down, one

can see that up to 8 threads P8TM commits ∼80% of the transactions as either HTM or ROTs. Moreover, unlike *intruder* where HTM-SGL was able to commit a smaller percentage of transactions in hardware, HTM-SGL is unable to scale with *yada*. This can be related to the difference in the nature of workloads, where the transactions that trigger capacity abort form the critical path of execution; hence with such workloads it is not preferable to use hardware-based solutions.

3.7.4 TPC-C Benchmark

Figure 3.12 shows the results for workloads with 10%, 50%, and 90% update transactions that consists of a mix of the five types of transactions of the port of TPC-C to the TM domain (see Section 2.4).

Throughput results show clear advantage of P8TM over all the other baselines in all workloads, regardless of the number of active threads. When compared with software based solutions, P8TM is able to achieve up to 5× higher throughput than both NOrec and HyNOrec at 16 threads in the 90% update workload. Although both NOrec and HyNOrec can scale up to 16 threads, their lower performance can be explained by the much lower instrumentation overheads that P8TM incurs when compared to software-based solutions. When compared to HTM-SGL, P8TM achieves 5.5× higher throughput with workloads that have a high percentage of read-only transactions, thanks to the URO path. When moving to workloads with higher percentages of update transactions, P8TM still outperforms HTM-SGL by 2× and 1.25× on the 50% and 90% update workloads, respectively. Again, looking at the breakdown plots, P8TM is able to commit all update transactions either as HTM or ROTs up to 8 threads. P8TM_{ucb} manages to achieve even further improvement in throughput by disabling the HTM path, hence decreasing the abort rate significantly.

3.8 Related Work

Hybrid TM [22, 23] (HyTM) attempts to address the efficiency of HTM's fallback issue by falling back to software-based TM (STM) implementations when transactions cannot successfully execute in hardware. However, state of the art HyTM solutions suffer from high synchronization costs to ensure correctness while executing HTM and STM concurrently.

HyNOrec aborts all HTM transactions when a STM transaction is committing even in absence of conflicts. RHyNOrec is only viable if the transaction's postfix, which may potentially encompass a large number of reads, does fit in hardware. Further, the technique used to enforce atomicity between the read-only and the remaining reads relies on instrumenting every read within the prefix hardware transaction, this utterly limits the capacity—and consequently the practicality—of these transactions. Unlike RHyNOrec, P8TM can execute read-only transactions of arbitrary length in a fully uninstrumented way. Further, the T2V mechanism employed by P8TM to validate update transactions relies on a much lighter and efficient read-set tracking and validation schemes that can even further increase the capacity of transactions. Other HyTM solutions such as HyTL2 and HyLSA require fulling

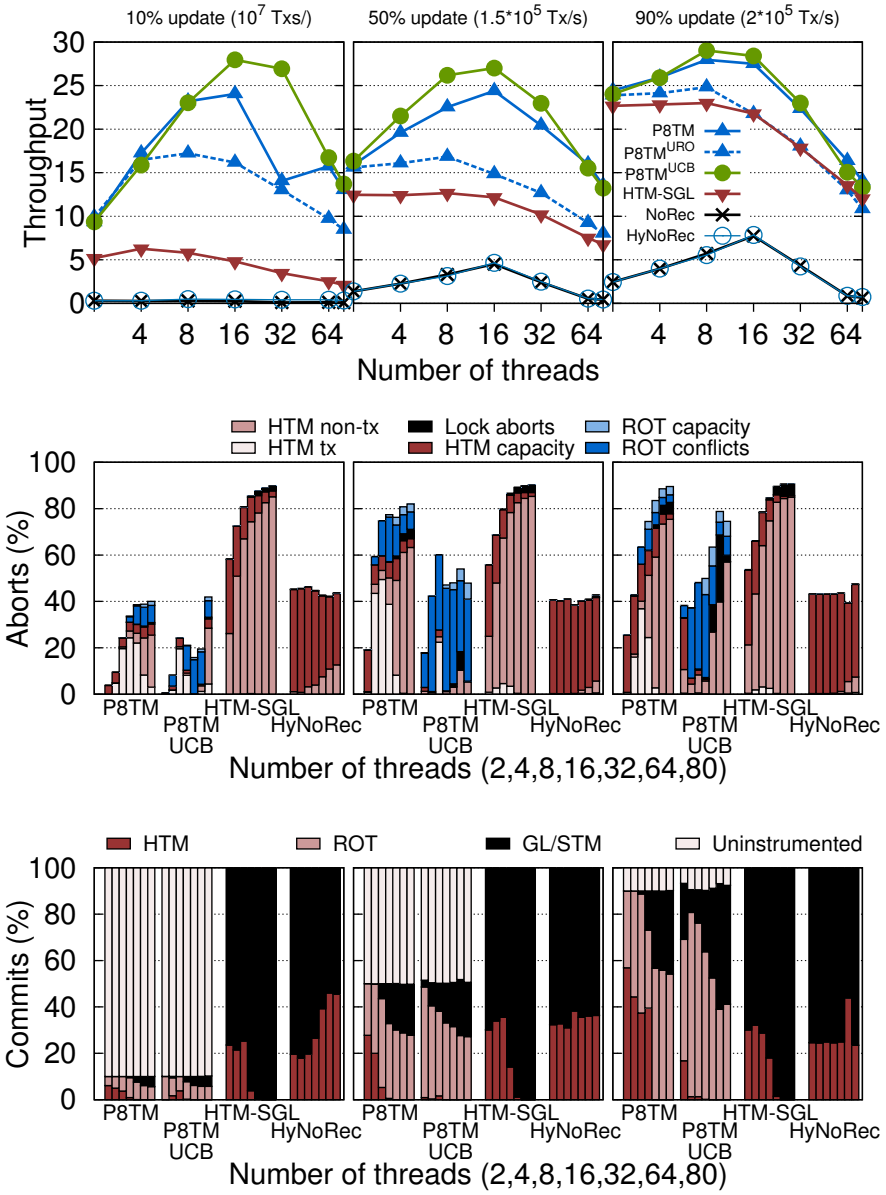

Figure 3.12: Throughput, abort rate, and breakdown of commit modes of TPCC at 10%, 50% and 90% update ratios.

instrumenting the writes only or both reads and writes respectively. This instrumentation further shrinks the capacity limitation of HTM unlike P8TM that strives to expand it.

P8TM is also related to the literature aimed to enhance HTM's performance by optimizing the management of the SGL fallback path. A simple, yet effective optimization, which is included in P8TM, is to avoid the, so called, *lemming effect* [69] by ensuring that

the SGL is free before starting a hardware transaction. An alternative solution to the same problem is the use of an auxiliary lock [113]. As these two solutions provide equivalent performance, P8TM integrates the former, simpler, approach. Calciu et al. [73] suggested lazy subscription of the SGL in order to decrease the vulnerability window of HTM transactions. However, this approach was shown to be unsafe in subtle scenarios that are hard to fix using automatic compiler-based techniques [114].

P8TM integrates a self-tuning approach that shares a common theoretical framework (the UCB reinforcement learning algorithm [110]) with Tuner [89]. However, Tuner addresses an orthogonal self-tuning problem to the one tackled by P8TM: Tuner exploits UCB to identify the optimal retry policy before falling back to the SGL path upon a capacity exception; in P8TM, conversely, UCB is to determine which synchronization to use (e.g., ROTs/UROs vs. plain HTM).

3.9 Summary

This chapter presented P8TM, a TM system that tackles one of the key limitations of existing HTM systems: the inability to execute transactions whose working sets that exceed the capacity of CPU caches. This is achieved via novel techniques that exploit hardware features available in the IBM POWER8 processor. P8TM was evaluated via an extensive experimental study, which highlighted the robustness of its performance across a wide range of benchmarks, ranging from simple data structures to complex applications, and achieves remarkable speedups.

The importance of P8TM stems from the consideration that the best-effort nature of current HTM implementations is not expected to change in the near future. Therefore, techniques, such as P8TM, that mitigate the intrinsic limitations of HTM can broaden its applicability to a wider range of real-life workloads. We conclude by arguing that the performance benefits achievable by P8TM thanks to the use of the ROT and S/R mechanisms represent a relevant motivation for integrating these features also in the future generations of HTM-enabled processors by other manufacturers.

Chapter 4

DMP-TM

By using in synergy hardware supports for TM and software-based mechanisms, the solution presented in the previous chapter, P8TM, expands the capacity limitation of HTM systems by an order of magnitude. Even so, the maximum capacity available for an update transaction in P8TM remains limited. In fact, as shown in the experimental evaluation of P8TM, there exist realistic workloads that exceed P8TM's capacity and that would benefit from the ability to execute long running update transactions in a pure STM-based path.

In this chapter, the focus shifts to a different type of TM, i.e. HyTM, which, unlike P8TM, use STM on the fallback path of HTM and can, as such, support the concurrent execution of update transactions that issue a (virtually) unlimited number of memory accesses. Unfortunately, though, as already mentioned (and confirmed in the experimental study of Chapter 3), the performance of state of the art HyTM systems is still far from fulfilling the promise of delivering the best of STM and HTM.

This chapter presents a novel HyTM algorithm, called Dynamic Memory Partitioning-TM (DMP-TM), that leverages operating system-level memory protection mechanisms in order to avoid the overheads incurred by existing HyTM systems to detect conflicts between HTM and STM transactions. Thanks to this design approach, DMP-TM is capable of supporting highly scalable STM implementations, without any instrumentation on the HTM path. Some passages in this chapter have been quoted verbatim from [43],

4.1 Problem

HyTM systems seek to obtain the best of STM and HTM by allowing HTM transactions to use some STM implementation as its fallback path. Unfortunately, despite the number of papers published in this area in recent years [24, 28, 115, 116], existing HyTM implementations still suffer from large synchronization overheads to ensure correctness when HTM and STM run concurrently [29]. State of the art HyTM systems trade-off either concurrency between HTM and STM for low instrumentation costs on HTM or efficiency of HTM for concurrency between both back-ends. As a result, state of the art HyTM systems actually

perform worse than HTM and STM in a wide range of workloads [70].

For example, HyTM systems that fallback to the LSA's STM [64] require HTM transactions to manipulate per-location metadata (often referred to as Ownership Records, *orecs*) used by STM transactions. This extends significantly the memory footprint of HTM transactions, making them prone to capacity aborts. The only HyTM solution that avoids instrumenting read and write memory accesses is HyNOrec (see Section 2.2.6), which uses the NOrec STM on its fallback path. However, NOrec is optimized for running at low threads counts and is known to be way less scalable than *orec*-based approaches, thus representing a suboptimal fallback path for large scale parallel systems. Further, HyNOrec induces *spurious aborts* of HTM transactions in presence of concurrent commits of non-conflicting STM transactions.

4.2 Overview

The key novel idea exploited in DMP-TM is to rely on operating system (OS) level memory protection mechanisms to detect conflicts between HTM and STM transactions. This design brings two important benefits, but also non-trivial challenges. A first key advantage is that DMP-TM is agnostic to the actual STM implementation being used: this allows DMP-TM to be used in conjunction with highly scalable and efficient *orec*-based STM systems, while avoiding the harsh instrumentation overheads imposed to the HTM path by existing HyTM systems. Another major benefit stemming from this design is that DMP-TM allows HTM and STM transactions that access disjoint memory pages to commit concurrently, sparing them from spurious aborts that would instead arise with state of the art HyTM systems [24]. The main mechanisms that compose the DMP-TM system are overviewed in the following.

Double-heap approach. Section 4.3.1 explains how DMP-TM maps the heap of a TM application twice in the process virtual address space, one view being accessed by the HTM path and one by the STM back-end. It then relies on OS memory protection mechanisms to selectively prevent pages of one heap from being accessed by the opposite back-end.

Dynamic memory partitioning. Section 4.3.2 explains how DMP-TM utilizes systems calls to enforce memory partitions during run-time. Mitigating the high cost of these systems calls is the key challenge of DMP-TM's design. Indeed, DMP-TM investigates an interesting trade-off: leveraging the data partitionability present in applications in order to reduce the runtime overheads of detecting conflicts among STM and HTM transactions, at the cost of a performance penalty in case conflicts between STM and HTM transactions do materialize.

Self-tuning. In order to maximize the gains achievable in favorable workloads, DMP-TM integrates lightweight mechanisms, described in Section 4.3.3, that detect, in a transparent and automatic way, which back-end (STM or HTM) to employ for the different transactional

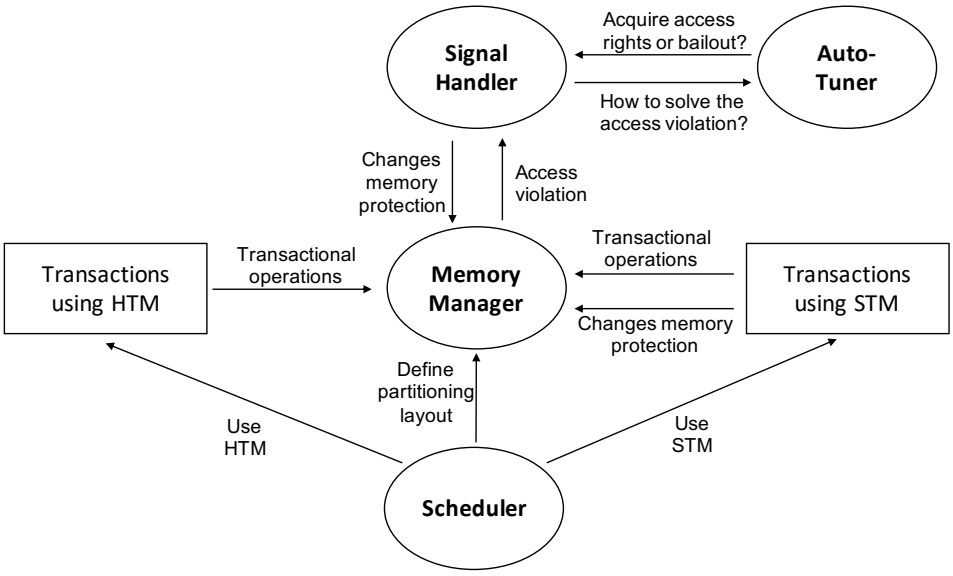

Figure 4.1: Architecture of DMP-TM.

blocks of a TM application. Moreover, DMP-TM makes use of lightweight heuristics also to automatically detect unfavorable workloads and ensure robust performance by falling back to use exclusively the most efficient of the two back-ends.

Section 4.5 shows an extensive evaluation study of DMP-TM based on both synthetic and realistic benchmarks, namely STAMP and the TPC-C [103] to the TM domain, and compared against HTM, two STM and three HyTM systems. The results of the study show that, in ideal workloads, DMP-TM achieves up to 8.1×/19× speedups vs the best STM/HyTM implementation and up to 37× vs HTM. DMP-TM achieves significant performance gains even when faced with realistic applications: DMP-TM outperforms all the considered baselines in two out of the three benchmarks of the STAMP suite that are favorable for HyTM systems, achieving up to 2× speedups versus the best alternative. Analogous speedups are obtained even with TPC-C. Overall, the study shows that DMP-TM can achieve significant performance gains even when faced with realistic workloads that do not exhibit perfectly partitionable access patterns, providing experimental evidence in support of the practical viability of the proposed solution.

4.3 Description

Figure 4.1 depicts DMP-TM's architecture, which is composed by four main components: *(i)* Memory Manager, *(ii)* Scheduler, *(iii)* Signal Handler and *(iv)* Auto-Tuner. It should be noted that these are 4 logical components, which, as detailed in the next sections, are physically implemented in a scalable and decentralized fashion in DMP-TM to enhance efficiency.

The Memory Manager is responsible for regulating the accesses by the HTM and STM paths to the shared transactional heap. This entails: mapping the transactional heap in the process address space twice (via the *mmap*() system call), granting and revoking access grants of the two paths to the shared heaps (via the *mprotect*() system call), as well as caching in user space the state of the per-page memory protections (an optimization that spares from executing system calls in absence of contention between HTM and STM).

The Scheduler classifies transactional blocks as either HTM-friendly or non-HTM-friendly, and accordingly establishes the execution paths to be used to support their execution.

The Signal Handler is activated when HTM transactions access a memory page that was last accessed in an incompatible mode by a STM transaction, triggering an access violation. In such a case, the Memory Manager is consulted to determine when it is safe to restore the needed protection for HTM.

Finally, the Auto-Tuner determines when it is beneficial to turn off one of the back-ends, as the excessively high frequency of system calls' activation outweighs the gains achievable by using concurrently the HTM and STM paths.

As already mentioned, DMP-TM is designed to be STM agnostic, i.e., it can enable concurrent execution of HTM with any STM algorithm. The flexibility enabled by this feature of DMP-TM is particularly relevant given that various works have shown the lack of a no-one-size-fits-all solution when it comes to STM systems [99]. In fact, DMP-TM's current prototype integrates TinySTM, which was selected due to its high scalability and to the efficiency of its implementation [70]. However, it would be straightforward to develop variants of DMP-TM using alternative STM implementations.

As for the HTM implementation, DMP-TM assumes a conventional/plain interface for transaction demarcation. The only assumption that DMP-TM makes on the underlying HTM system is that, upon a page access violation, the HTM implementation aborts the transaction and makes available to the signal handler the information on the address that triggered the exception. This is an information that HTM implementations by Intel do not currently disclose, but that is instead provided by IBM POWER8's HTM (and by other IBM implementations, to the best of our knowledge).

4.3.1 Memory Manager: Double Heap Approach

DMP-TM manipulates the process' virtual space in such a way that the heap is mapped twice (see figure 4.2) — one heap being used by STM transactions (STM heap) and the other by HTM transactions (HTM heap). Although both heaps point to the same data, with this arrangement, it is feasible to control the access rights of specific regions in one of the heaps, without affecting the other heap. It is worth noting here that this solution does not restrict a transaction type to be executed by a specific thread, i.e., any thread can execute any transaction, either using HTM or STM, at any moment of time. DMP-TM automatically maps the access to shared data to the correct heap according to the back-end being used to execute a given transaction.

Upon its initialization, DMP-TM creates a shared memory zone using the directive

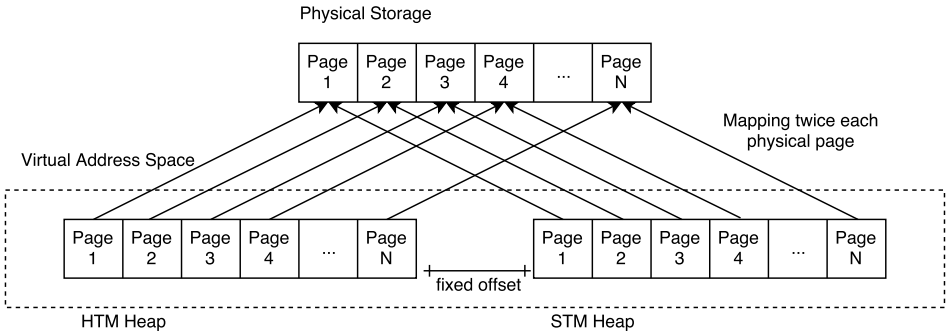

Figure 4.2: Mapping of the address space in the HTM Heap and in the STM Heap.

shm_open(). Then, using the Unix system call *mmap()* the shared memory zone is mapped twice to the HTM heap and STM heap. To control access rights, DMP-TM uses the *mprotect()* system call, which operates at the granularity of a single page. This system call changes the protection of memory pages contained in a given range of addresses. In order to allow an efficient way to calculate the page to be revoked in the opposite heap, both heaps are placed at a constant offset. Thus, calculating the address in the opposite heap is achieved by simply adding or subtracting a fixed offset. This implies basically two changes in the way applications should be developed to be used with DMP-TM:

1. Dynamic memory should only allocate memory from the shared memory region. To this end, DMP-TM integrates a simple, custom implementation of malloc that allocates memory exclusively from the range of addresses associated with the shared region. Analogously to other memory allocators, e.g., [32], DMP-TM's custom malloc implementation splits the range of virtual addresses in n equally sized, disjoint, page aligned splits, where n is the number of threads. This allows each thread to serve malloc/free requests from its "private" split, avoiding any synchronization overheads with other threads.

2. Two code paths need to be produced, one for STM and one for HTM transactions, each targeting the corresponding heap. Given that the translation between the two address spaces is simple (just a plain translation), the generation of the code paths could be fully automated by a compiler — although the current prototype of DMP-TM does not provide compiler support for this task.

4.3.2 Memory Manager: Enforcing Dynamic Partitions

DMP-TM spares HTM transactions from having to check or notify the STM path about possible conflicts, placing any required instrumentation on the STM path. Throughout the execution of an STM transaction, before any shared data access, DMP-TM checks if that data lies on a page accessible, in an incompatible mode, by HTM and, if needed, it accordingly removes the access permissions from the corresponding page in the HTM heap.

The result of this design is that whenever a HTM transaction accesses a page for which it does not own adequate access rights, the OS (which, in its turn, exploits virtual memory hardware supports) triggers an access violation by raising a *SIGSEGV* signal. This causes the immediate abort of any HTM transaction that has already accessed that page, or that will access that page in the future. This signal is treated by the Signal Handler module, which is in charge of restoring access rights for HTM transactions.

In order to regulate access to the HTM and STM heaps, DMP-TM stores the following per page metadata:

- *Status field*, which tracks the access rights of a page in the HTM heap. Pages can be in one of three states: *(i)* Read/Write: HTM can update data on this page, *(ii)* Read: HTM can only read data from this page and *(iii)* None: HTM cannot access this page in any form. This allows the STM to retrieve in an efficient way, i.e., without issuing system calls, the access permissions of the pages in the HTM heap.

- *Transition count*, which stores how many times the write access permissions to a page have been restored by the Signal Handler. This counter is monitored by STM transactions to detect if the write permissions to a previously read page have, in the meanwhile, been granted back to HTM. If this is the case, some HTM transaction may have overwritten a value previously read by the STM transaction, which is, thus, restarted.

- *Writers Count*, which tracks the number of active STM transactions that wrote to a page. This counter is atomically incremented by an STM transaction upon its first write to a page and atomically decremented upon its commit or abort. This variable is used to prevent restoring access rights to a page in the HTM heap, while there are active STM transactions that wrote to it, thus, preventing HTM transactions from observing inconsistent states.

- *Lock bit*, which acts as a mutex that is acquired whenever the protection and state of a page have to be altered, preventing transactions from concurrently altering the state and protection (via *mprotect()*) of the same page.

In fact, DMP-TM strives to take advantage of the partitionability of the memory access patterns generated by TM applications, a property that was already observed in some reference TM benchmarks in previous works [117] and that is also confirmed by the experiments in Section 4.5. When TM applications do exhibit such a property, DMP-TM allows both HTM and STM transactions to execute avoiding mutual interference, thus minimizing the synchronization of STM transactions and completely removing the need of synchronization overheads for the HTM side. With workloads that generate excessive contention between HTM and STM transactions, the cost of migrating page protections from a heap to the other may outweigh the performance gains stemming from the avoidance of expensive synchronization mechanisms in non-contended runs.

4.3.3 Transaction Scheduler and Auto-Tuner

As discussed earlier, the Scheduler module has the responsibility of determining the back-end (HTM or STM) to be used by each transaction. DMP-TM utilizes simple heuristic to accomplish this task. The Scheduler tracks the number of aborts due to the exceeding of the cache capacity by HTM transactions. If the ratio (*capacity aborts/number of commits+capacity aborts*) is greater than 90%, this transaction type is labeled as STM.

In workloads with high degrees of contention between the HTM and STM back-ends, DMP-TM is likely to incur high costs due to the cost of handling access violations and issuing system calls to restore the access rights on the HTM heap. In order to detect when these costs outweigh the gains of executing HTM without instrumentation concurrently with STM, the Auto-Tuner module of DMP-TM employs a *bailout* mechanism based on the following heuristic: If DMP-TM is spending more than 20% of time issuing system calls, it resorts to using either of the back-ends. The decision of upon which back-end to fallback to is taken by sampling the throughput of both back-ends and choosing the back-end with higher throughput.

4.4 Algorithm

Algorithms 6 and 7 show the pseudocode for the STM path and the Signal Handler's logic respectively. To simplify presentation, the pseudocode only presents the core functionality that allows STM and HTM transactions to correctly execute concurrently, omitting the logic of the Scheduler and Auto-Tuner (see Sec. 4.3.3), as well as several optimizations that are discussed in Sec. 4.4.2.

STM reads. When a read is issued by a STM transaction, it is first checked if the `transition_count` of previously read pages has changed since the last access. If any of them has changed in the meanwhile, it means that some HTM transaction may have updated that page (and possibly committed); thus, the STM transaction is aborted. Next, if it is the first read access to this page by this transaction, it stores a local copy of the `transition_count` to use it for future checks. Then, it checks the metadata of the page to which it is issuing a read. If the page's access rights are Read/Write (Line 11), which means that a HTM transaction can perform updates on it, then the `lock_bit` is acquired in order to change the protection of the page to Read, allowing concurrency with HTM transactions that read this page. After setting the metadata of the page and releasing the `lock_bit`, the transaction effectively reads the memory position, using the underlying STM's API, and checks again if the `transition_count` has changed. If so, the performed read is not legal and consequently, the transaction is restarted. If not, the read is successful.

STM writes. Upon a write to shared data from within a STM transaction, if it is the first time the transaction writes to a page it atomically increases the `writers_count` for that page (Line 23). This blocks any attempt by concurrent Signal Handlers of changing the HTM access rights for that page. Then it is checked if the page corresponding to the location to be updated is accessible by HTM (Line 25). If so, after acquiring the `lock_bit`,

Algorithm 6 DMP-TM: pseudocode for STM

```
1: Shared variables:
2:     status[N], tc[N], wc[N], lb[N] ← {0, 0, ..., 0}
                          ▷ per page variables for status field, transition count, writers count and lock bit
3: Local variables:                                          ▷ initialized upon every transaction attempt
4:     private_tc[N] ← {0, 0, ..., 0}                        ▷ local version of tc
5:     pages_read, pages_writter ← ∅                         ▷ set of pages accessed as read and write
6: function STM_READ(addr)                                  ▷ get page where this address lies
7:     VALIDATE_READSET()
8:     if page ∉ pages_read then                             ▷ First time page is read
9:         pages_read ← pages_read ∪ page
10:        private_tc[page] ← tc[page]
11:    if status[page] == #READWRITE then
12:        ACQUIRE lb[page]
13:        MPROTECT(READ)                                    ▷ issue mprotect with read-only
14:        status[page] ←#READ                               ▷ set status to read
15:        RELEASE lb[page]
16:    val ← TX_Read(addr)                                   ▷ call the STM API
17:    if private_tc[page] ≠ tc[page] then
18:        STM_RESTART()
19:    return val
20: end function
21: function STM_WRITE(addr,val)
22:    if page ∉ pages_written then                          ▷ First time page is written
23:        ATOMIC_INCREMENT(wc[page])
24:        pages_written ← pages_written ∪ page
25:    if status[page] ≠#NONE then
26:        ACQUIRE lb[page]
27:        MPROTECT(NONE)                                    ▷ issue mprotect with none
28:        status[page] ←#NONE                               ▷ set status to none
29:        RELEASE lb[page]
30:    TX_Write(addr, val)                                   ▷ call the STM API
31: end function
32: function VALIDATE_READSET
33:    for page ∈ pages_read do
34:        if tc[page] ≠ private_tc[page] then
35:            STM_RESTART()
36: end function
37: function STM_RESTART
38:    for page ∈ pages_written do
39:        ATOMIC_DECREMENT(wc[page])
40:    TX_Abort
41: end function
42: function STM_COMMIT
43:    VALIDATE_READSET()
44:    TX_Commit                                             ▷ ask the STM to commit
45:    for page ∈ pages_written do
46:        ATOMIC_DECREMENT(wc[page])
47: end function
```

the access rights for HTM are revoked. This protects HTM from witnessing inconsistent states by observing values written by incomplete STM transactions. After changing the page access rights via *mprotect()* to None and updating the *status field* of the page, the `lock_bit` is released and the transactional update of the value is finally performed.

STM aborts. Before a STM transaction aborts, either due to data conflict or abort from DMP-TM, the `writers_count` of all the pages previously written by the transaction is

Algorithm 7 DMP-TM: pseudocode for signal handler

```
1: function HANDLE_AV(addr,isReadOnly)
2:     wait until wc[page] = 0                          ▷ drain writing STM transactions
3:     Acquire lb[page]
4:     if ¬isReadOnly then
5:         status[page] ←#READWRITE
6:     else
7:         status[page] ←#READ
8:     mem_fence
9:     if wc[page] ≠ 0 then
10:        status[page] ←#NONE
11:        Release lb[page]
12:        go to 2                                      ▷ wait for writers count to be 0
13:    if ¬isReadOnly then
14:        tc[page] + +
15:        mprotect(Read/Write)
16:    else
17:        mprotect(Read)
18:    Release lb[page]
19: end function
```

decreased (Lines 38-39). This allows HTM transactions to regain accesses to those pages.

STM commits. After a STM transaction finishes its execution, it enters in the commit phase, in which it first checks if the `transition_count` of the pages previously read have changed meanwhile (Line 43). In the case they have changed, the transaction restarts as explained before. Otherwise, it continues to commit according to its implementation-dependent logic. Then, finally, it decrements the `writers_count` atomically for all the pages it has written to (Lines 45 - 46).

Signal Handler. The Signal Handler (function HANDLE_AV()) is activated whenever a HTM transaction accesses a page for which it does not have adequate access rights. In this case the OS generates a SIGSEGV, which is intercepted and managed in the same thread that generated the exception. After having extracted the target address of the memory operation that triggered the access violation[1], the Signal Handler waits for the *writers count* of the corresponding page to be zero. Next, it acquires the page's `lock_bit`, sets the metadata to Read/Write or Read (depending on whether the exception was generated in an update or a read-only transaction) and checks again for the `writers_count`, to ensure that it did not change in the meanwhile. Otherwise, the Signal Handler defers to any active writing STM by resetting the access rights to None and going back to wait until the `writers_count` is zero. After that, the Signal Handler can restore the HTM access rights to the desired page and, in case the exception occurred in an update transaction, it increments the `transition_count` to notify STM transactions about the occurrence of possible conflicts with HTM transactions. It should be noted that `transition_count` is increased before acquiring the Read/Write rights, in order to guarantee that if a HTM

[1]This information is obtained via the *siginfo* struct that is passed by the OS to the Signal Handler. Existing Intel implementations of HTM reset the siginfo if the access violation occurs in a hardware transaction, which is the reason why DMP-TM does not currently support Intel's architecture.

transaction is granted back permission to update and commit a page (via MPROTECT()), the STM path is guaranteed to detect the corresponding change of the transition count.

4.4.1 Correctness Argument

This section provides a set of (informal) arguments on the correctness of the DMP-TM. The analysis is organized by discussing, separately, how DMP-TM enforces isolation of HTM and STM transactions.

Isolation of HTM transactions. The key invariant enforced by DMP-TM to ensure correctness of a HTM transaction T_{HTM}, despite the concurrent execution of any STM transaction T_{STM}, is to ensure that T_{HTM} has no permission to access any of the pages written by T_{STM} throughout its execution. To this end, STM transactions remove HTM's access rights to each page they write to, before issuing the actual write operation. As already mentioned, the removal of the access right causes the immediate abort of any HTM transaction that had already read/written that page, as well as future accesses by HTM transactions to that page.

It is however necessary to carefully synchronize the concurrent execution of the Signal Handler(s) and STM transactions that compete to restore/remove the access rights of the same page. In particular, it is necessary to ensure that the Signal Handler can restore HTM's access to a page only when there are no active STM transactions updating that page, i.e., when the page's `writers_count` is set to zero. This is achieved by having the Signal Handler check for the `writers_count` twice, while setting the metadata to Read/Write in between. For the second check to be valid, there can be no concurrent STM transaction that started a write operation on this page yet — recall that the `writers_count` is atomically incremented as the first step of processing a STM write. In case of a STM transaction T starting a write operation after the second check of the Signal Handler, then T will notice that HTM transactions have access to the page, thanks to the memory barrier that precedes the second check (Line 8); in this case, T will acquire the page's lock (synchronizing with any concurrent Signal Handler operating on the same page) and issue a MPROTECT that will abort any concurrent HTM transaction.

Isolation of STM transactions. In order to ensure correctness of STM transactions, DMP-TM guarantees that none of the pages accessed by STM transactions can be altered by HTM transactions, since the time in which each page is first read and until the end of the STM transaction. This is achieved via two key mechanisms: i) ensuring that a STM transaction accesses a page after having removed any non-compatible permission to the corresponding HTM heap's page; ii) checking the transition count of every read page, upon each read and before commit, letting the transaction proceed (without aborting) only if none of them changed since the first time in which that page was read. This implies that the page was not concurrently modified by a HTM transaction, since the `transition_count` would be found different if protections had been changed in the meanwhile. It should also be noted that the atomicity of each individual STM read is guaranteed by reading the `transition_count` of a page before and after performing the read via the API of the underlying STM implementation.

It should be noted that, in order to reduce overheads, STM transactions check the status of a page without first acquiring the corresponding lock. It is hence possible that a STM transaction finds, in Line 11, a page as not writeable by the HTM path and that, before it completes the read, a Signal Handler restores write permission to HTM for the same page. In this case, HTM transactions may even commit and update to that page, before the STM completes executing its read. In such a case, though, the transition count would be found to have changed, when it is checked for the second time in Line 17 by the STM transaction. If, instead, the values of the transition counts are not found to have changed, then the STM read is also guaranteed to observe the permissions set by the Signal Handler when it increased the value of transition count — as the memory fence in Line 8 ensures that if the increase to the transition count is globally visible, so is the corresponding page's state. This causes the STM transaction to synchronize with concurrent Signal Handlers, by acquiring the page's lock, and to ensure that HTM write permissions are removed before performing the read.

4.4.2 Optimizations

The following are a set of optimizations that can be applied to the algorithm described above to further enhance its performance:

- instead of checking the `transition_count` of every accessed page upon each access, one can use a global `transition_count` that is incremented atomically from within the Signal Handler together with the increment of the page's `transition_count`. Then instead of checking each `transition_count` of every page upon each STM access, it suffices to only check if the global `transition_count` has changed. If it has not (the common case in workloads that exhibit good partitionability), no further checks are required; else, the normal procedure, where the STM transaction checks the `transition_count` of each page it previously read, takes place.

- instead of maintaining a single *writers count* per page shared by all STM threads, which must be incremented or decremented atomically, one can use a set of, per thread, local counters. This will spare STM transactions from the need to perform expensive atomic operations. However, the Signal Handler will need to ensure that all the flags are unset before it attempts to change the page's protection.

- certain STM algorithms need to re-validate their read-set in order to ensure the safe execution of transactions, e.g., as in the case of TinySTM in presence of concurrent commits of update transactions. These STM implementations could be made aware of the execution of read-set validations performed by DMP-TM (if `transition_count` is found to have increased), which may allow them to spare duplicate validations.

The current implementation of DMP-TM integrates the first two optimizations, but not the last one. Unlike the first two mechanisms, in fact, the last optimization comes at the cost of requiring to alter the inner logic of the STM implementation employed by DMP-TM, whereas one of the key design goals of DMP-TM is its STM-agnostic nature.

4.5 Evaluation

This section reports the results of an extensive experimental study, in which both the static (DMP-TM) and the dynamic version (DMP-TUNE) of the proposed solution were evaluated. In the static version, transactions are statically assigned to either one of the back-ends according to an exhaustive offline search for the best configuration, whereas in the dynamic version the scheduler module decides upon the transaction assignment during run-time. DMP-TM and DMP-TUNE were compared against: *(i)* HTM with a single global lock as fallback (HTM-SGL); *(ii)* TinySTM with the same configuration as the one used for DMP-TM and DMP-TUNE; *(iii)* HyNOrec using two counters to decouple subscribing from signaling; *(iv)* NOrec with write back configuration; v) HyTinySTM, which is implemented by adapting the original algorithm [25] to replace the *prefetchw* instruction, which is not available in current HTM implementations, with a write operation (see Section 2.2.6); *(vi)* HyTL2, based on the algorithm described in [26]. All hardware-based solutions try executing each transaction 10 times in hardware before resorting to the fallback path. DMP-TM and DMP-TUNE, however, try the transactions attributed to HTM 100 times before acquiring the global lock. This is done since the transactions attributed to HTM in DMP-TM and DMP-TUNE are hardware-friendly and likely to commit using HTM, if tried long enough. Conversely, using such a high retry count with other hardware-based solutions will dramatically degrade their performance, since the approaches try *all* transactions (including non-HTM-friendly ones) first in hardware.

This section starts by using synthetic benchmarks to generate diverse workloads. intended to test extreme scenarios regarding partitionability of HTM and STM access patterns. Then, DMP-TM is tested using real-world complex benchmarks, namely two benchmarks of the popular TM benchmark suite STAMP [101] and TPC-C [103]. All presented results were obtained by executing on an 80-way IBM POWER8 8284-22A processor with 10 physical cores, where each core can execute 8 hardware threads. The OS installed is Fedora 24 with Linux 4.7.4 with page size of 64KB and the compiler used is GCC 6.2.1 with -O2. The reported results are the average of 5 runs.

Thread pinning was used to pin a thread per core at the beginning of each run for all the solutions until exhausting the number of available cores, and then distributing them in round-robin fashion to minimize unbalances between cores.

4.5.1 Synthetic Benchmarks

In order to assess the effectiveness of DMP-TM in diverse, yet identifiable workload settings, a synthetic benchmark based on two different hashmaps, storing 128, resp. 1024, elements per bucket was used. This setting was motivated by the fact that if HTM transactions read all the elements of a bucket from the first hashmap, the size of the read-set fits in the cache. However, if HTM transactions read all the elements from the second hashmap, the read-set will exceed the cache size, thus causing a capacity abort.

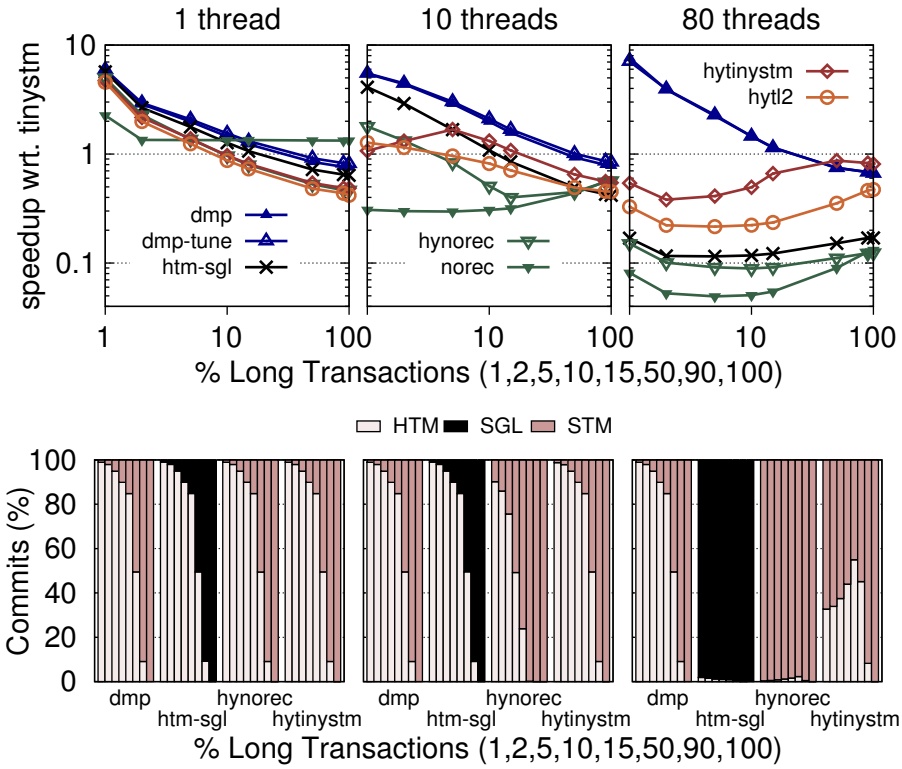

Figure 4.3: Speedup and commits breakdown for disjoint data structures running 1, 10 and 80 threads.

Disjoint data structures

To demonstrate the potential of DMP-TM, a workload, composed of two transactions types was used. One amenable for execution in hardware and one not (as it exceeds deterministically HTM's capacity). The workload generates perfectly partitionable memory accesses, i.e., the sets of pages accessed by each transaction type are disjoint. Both hashmaps are populated to use 64 pages in total and used a workload with 10% lookups and 90% update transactions.

Figure 4.3 reports the throughput speedup normalized with respect to TinySTM and the breakdown of commits for three different thread configurations: 1, 10 and 80 threads. For each of these configurations, the percentage of small transactions executed was varied. The x-axis reports the probability of a thread to be executing a long transaction from 1% to 100% (only long transactions). The results show remarkable gains either for DMP-TM and DMP-TUNE, since in this experiment both data structures are disjoint and the operations executed are so heterogeneous that HTM shines when executing short transactions, whereas STM excels with long ones. The throughput of all the solutions is normalized according to TinySTM (which is accordingly omitted from the plot), and use log scale on both y and

x-axis to enhance visualization.

With one thread, the overhead that DMP-TM and DMP-TUNE incur can be assessed. When the workload constitutes mainly small transactions, both variants of DMP-TM achieve better performance than HTM-SGL, thanks to their ability to run HTM transactions without any instrumentation and executing large transactions in STM — which spares the cost of retrying them several times before using the fallback path. As the percentage of large transactions in the workload increases, DMP-TM variants start to be outperformed by TinySTM, paying a penalty of ∼20% in the 100% long transactions workload. This is the cost of the extra instrumentation that DMP-TM adds on top of TinySTM. Note that NOrec consistently outperforms TinySTM, thanks to its more lightweight instrumentation.

At 10 threads, DMP-TM becomes the best performing back-end, achieving speedups of up to ∼2× compared to HTM-SGL, and more than 3× compared to TinySTM and ∼ 6× compared to NOrec-based solutions. This can be explained by the breakdown of commits shown in the middle row of Figure 4.3. DMP-TM is able to execute short transactions in HTM and long transactions as STM, unlike HTM-SGL that executes large transactions using the pessimistic single global lock. In this configuration, TinySTM starts to surpass the throughput of HTM-SGL when the workload is running operations in the smaller data structure with probability less than 20% but only at 50%, TinySTM's throughput equalizes DMP-TM. At 100% of large transactions, due to the fact that DMP-TM requires additional instrumentation on top of STM, it suffers ∼20% performance penalty w.r.t. TinySTM.

80 Threads continue showing the same trend as 10 threads, but with even greater speedups: more than 20× compared to HTM-SGL, ∼10× compared to HyTinySTM and ∼7× compared to TinySTM. The gains with respect to both HTM-SGL and HyTinySTM are due to DMP-TM's ability to execute more transactions in hardware, as shown in the commits breakdown plot. At high number of threads, capacity aborts become non-deterministic as more threads share hardware resources and it is, thus, beneficial to retry more times in hardware than reverting to the fallback path. However, without differentiating between deterministic and non-deterministic capacity aborts, high retry counts becomes harmful in terms of throughput. Again, TinySTM outperforms DMP-TM whenever the probability of executing large transactions is greater than 30% leading to a 30% overhead at 100% large transactions.

Throughout the entire experiment, the dynamic version of the algorithm (DMP-TUNE) worked as expected, since whenever transactions execute operations in the smaller hashmap the percentage of aborts due to exceeding transactions footprint is negligible. Nevertheless, when transactions execute operations in the larger hashmap, they deterministically abort due to exceeding the cache size. Thus, the scheduler module assigns the type of transactions accessing large hashmap as STM transactions, matching the offline assignment used by DMP-TM. Thanks to this assignment ability, DMP-TUNE is able to still benefit from using high retry counts unlike other hardware-based solutions.

Non-disjoint data structures

This experiment stresses the worst case scenario for DMP-TM, by allocating both hashmaps in the same memory region and interleaving the buckets of each hash map. With the granularity of DMP-TM being a single page, any access to either of the hashmaps is going to be considered a conflict. Therefore, DMP-TM will suffer from a very large number of system calls, as for either of the back-ends to commit a transaction, it is most likely that some page protection has to be restored, given that a 90% update workload is being considered.

Figure 4.4 depicts the results of running this workload with 1, 10 and 80 threads. It is clear that DMP-TM suffers large performance penalties, up to 20× compared with TinySTM in the worst case. This is true across all workloads except when the workload is dominated by either short or long transactions. At those extremes, there is no need for changing the pages access rights since DMP-TM is executing only one of the back-ends. Other than that, DMP-TM incurs significant overheads. This can be explained by looking at the ratio of system calls to commits, which shows that DMP-TM can pay up to more than 0.8 system calls per commit.

This is a typical workload where the Auto-Tuner module should decide to *bailout* and stick to either one of the back-ends. By inspecting the speedup plots in Figure 4.4, it can be seen that DMP-TUNE manages to match the performance of the best performing of both back-ends. At 1 thread, HTM-SGL performs better than TinySTM, except when the system is running only large transactions (right side of the 1 thread figure); so, as expected, DMP-TUNE falls back to HTM-SGL with percentage of long transaction less than 100%. At 10 threads, by looking at the commit breakdown is possible to see that DMP-TUNE falls back to HTM-SGL in workloads characterized by less than 50% of long transactions. However, after this mark, TinySTM begins to be the best performing back-end. Thus, DMP-TM falls back to TinySTM at this mark. At 80 threads mark, the best performing back-end is TinySTM, independently of the percentage of long transactions, and DMP-TUNE correctly adapts itself to employing TinySTM.

4.5.2 STAMP Benchmark Suite

Out of the 8 applications in the STAMP suite (see Section 2.4), *genome* and *intruder* are the ones that typically benefit from HyTM systems as they encompass both small and large transactions. Other applications generate either only small transactions (*ssca2* and *kmeans*) or large ones (*labyrinth* and *yada*). Thus, there is no room for improvement for any HyTM. The remaining two applications are *bayes* and *vacation*. The *bayes* application is known to suffer of very high variance and yields unreliable results [112]. The *vacation* application can also benefit from a HyTM system, however, it has low degree of partitionability. Hence, DMP-TUNE would perform as good as either HTM or STM.

The **genome** application represents the process of reconstructing the original source genome from a pool of DNA segments. An extensive brute-force experimental study was conducted in order to infer which of the 5 transaction types generated by *genome* to run with

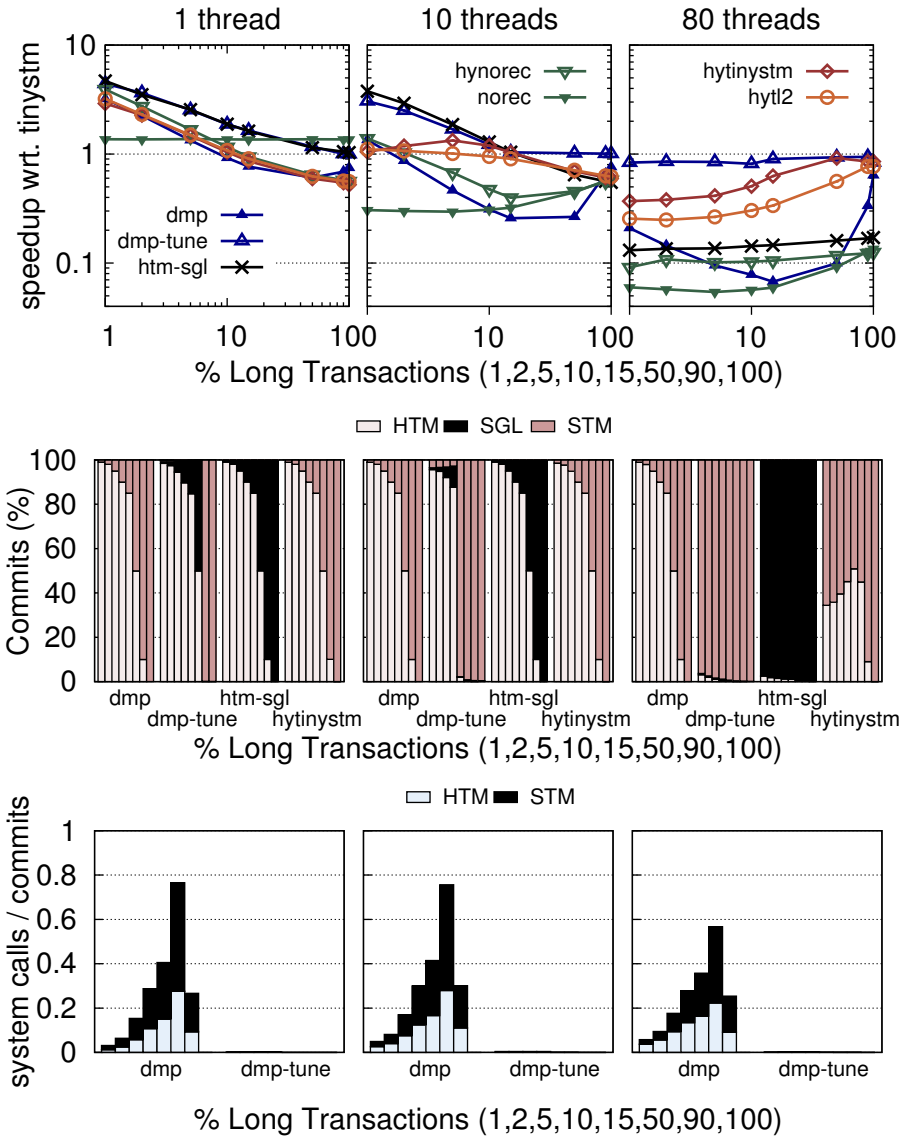

Figure 4.4: Speedup, commits breakdown and system calls ratio for non-disjoint data structure running 1, 10 and 80 threads.

HTM or STM. It was found that this workload indeed presents partitionability, as there are three disjoint transactional clusters according to their data access. DMP-TM successfully exploits this workload's property, achieving the maximum throughput and scaling to 64 threads, yielding ∼2× higher throughput than HyTinySTM, the second best baseline at 80 threads (Figure 4.5).

This can be explained by analyzing the commits breakdown, which shows DMP-TM's ability to execute >90% of transactions in hardware, ∼1% using the pessimistic fallback

path and ∼5% as STM at all thread counts. Although HyTinySTM manages to demonstrate similar commit patterns up to 8 threads, it performed worse than DMP-TM due to the extra instrumentation it imposes to its HTM path. Beyond 8 threads, HyTinySTM could not commit as many transactions in hardware, since it incurs more frequent capacity aborts that consume its retry count and lead to more frequent activations of the fallback path.

Furthermore, the workload is characterized by low contention. Therefore, up to 4 threads, NOrec and TinySTM achieve slightly better throughput than DMP-TM as they do not impose any extra instrumentation to their STM path. Up to 2 threads, HyNOrec, shows similar throughput as DMP-TM, since as shown in the commit breakdown plots of Figure 4.5, it still manages to commit 95% and 38% of the times in hardware, respectively for 1 and 2 threads, and uses as fallback NOrec, which as stated before achieves higher throughput than DMP-TM in this workload for a low thread count. However, as the thread count increases, abort rate starts to increase. In these contention settings, DMP-TM benefits from executing transactions in software, which enables more concurrency than the SGL fallback used by HTM-SGL. For the case of HyNOrec, the fallback of only one thread makes all the threads fallback to NOrec, and this has an adverse impact on performance when the thread count is higher than 4. After 16 threads, HyTinySTM starts to incur overheads due to the fact that at this thread count, cores are shared by more than one hardware thread, which reduces the effective cache capacity available for each hardware thread. As HyTinySTM instruments hardware transactions to check for changes in the STM *orecs*, it suffers from an increased abort rate compared to the solutions that do not instrument hardware transactions, namely HTM-SGL and DMP-TM. For the case of HyTL2, unlike the STM counterpart, TL2, not shown in the study, does not extend the snapshot used during STM reads, which leads to an increase of the transaction's abort rate.

The **intruder** application is a signature-based network intrusion detection system that encompasses three parallel transactions. The right column of figure 4.5 shows DMP-TM being the only back-end to scale up to 16 threads achieving ∼1.5× higher throughput than TinySTM, the second best performing back-end. The lower speedups in *intruder*, as compared with *genome*, can be attributed to the lower percentage of transactions (∼30%) that DMP-TM manages to execute in hardware. Again, HyTinySTM, which even commits more transactions in hardware, is outperformed by DMP-TM achieving 2× lower throughput at 16 threads, due to the costly instrumentation of its HTM path. NOrec's HyTM counterpart follows the same trend as NOrec. However, the commit breakdown shows that starting the execution in the HTM path and falling back to NOrec causes performance losses in the order of 0.73× compared to NOrec. It is worth noting that NOrec achieves the best throughput until 4 threads, thanks to its lower instrumentation costs. However, at 8 threads, its throughput deteriorates due to the increase of the contention in the workload. Further, HTM-SGL and HyTL2 are the worst back-ends, due to the pessimistic nature of the fallback of the former and to the inability of the STM fallback path of the latter to perform well in high contention workloads.

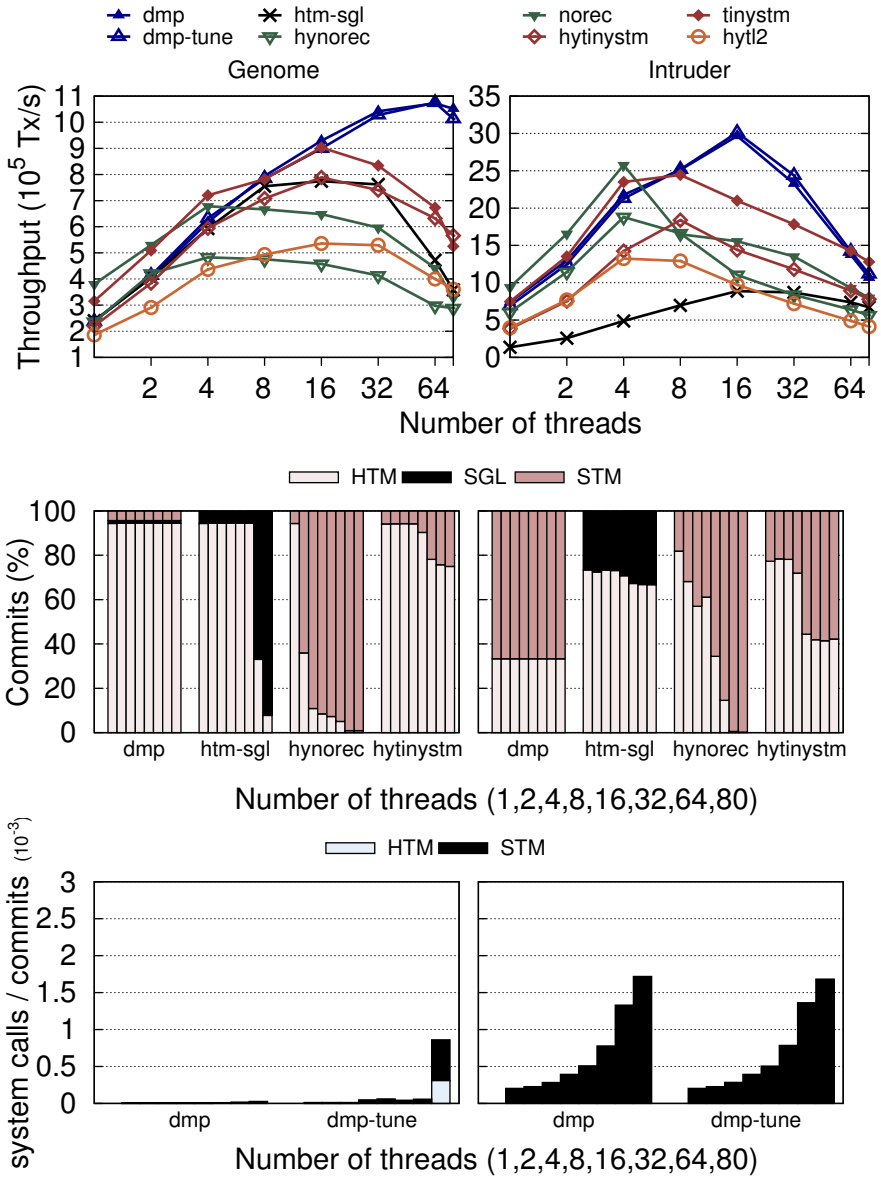

Figure 4.5: Speedup, commits breakdown and system calls ratio for *genome* and *intruder* of STAMP benchmark suite.

4.5.3 TPC-C Benchmark

Finally, DMP-TM is evaluated using a port of the TPC-C benchmark to the TM domain (see Section 2.4). To promote partitionability, vertical partitioning was performed according to the TPC-C standard, by moving attributes that are points of conflict to different memory regions to reduce false conflicts. Two different workloads that exhibit different degrees of

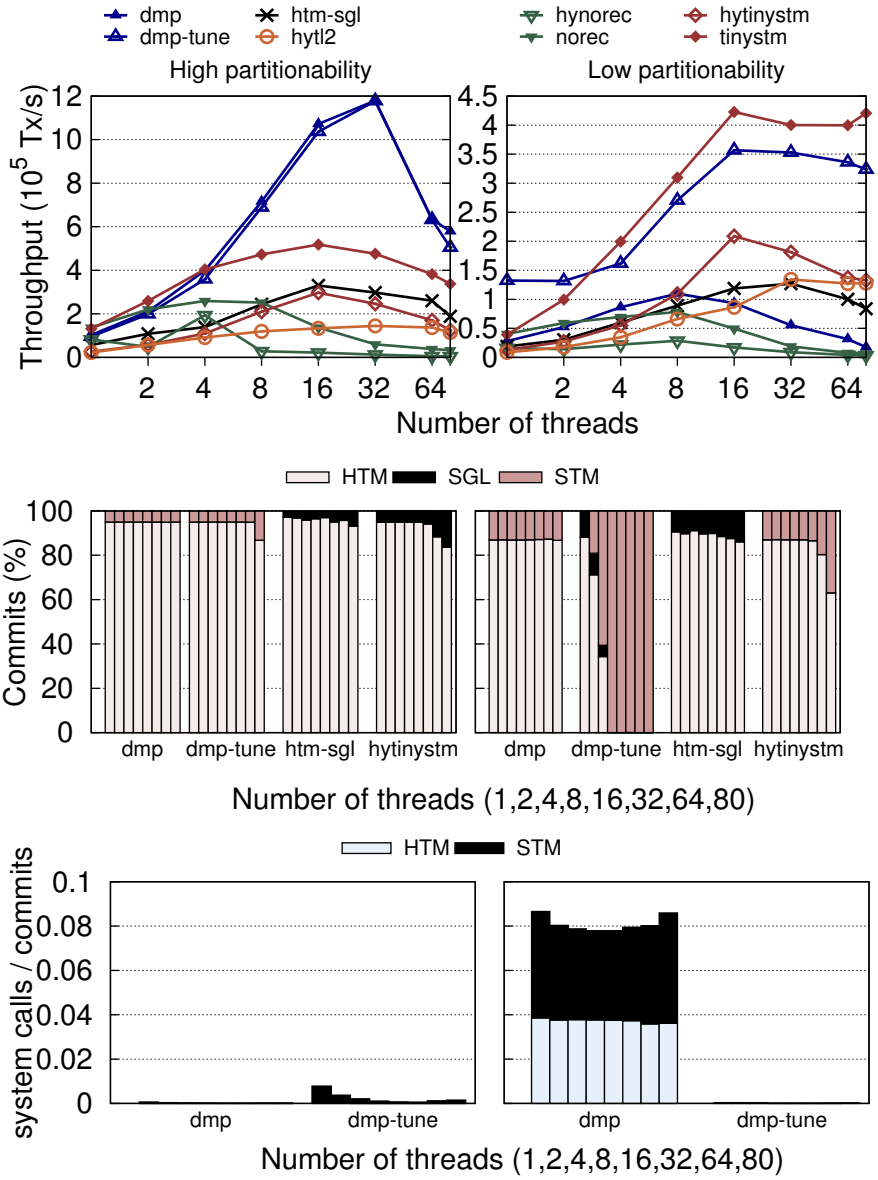

Figure 4.6: Speedup, commits breakdown and system calls ratio for two workloads of TPC-C.

partitionability between short and long transactions are considered.

The left column of Figure 4.6 reports the results of a workload composed by 95% of payment, 1% stock level and 4% of delivery transactions. This workload has a high degree of partitionability, reflected in a very low system calls to commits ratio. DMP-TM achieves the best throughput showing up to 2.4× speedups compared with the second best contender, TinySTM. At a low thread count, namely up to 4 threads, TinySTM

achieves slightly better throughput than DMP-TM due to the fact that it has no extra instrumentation. Although HTM-SGL and HyTinySTM execute more than 90% of the transactions in hardware, they yield ∼3× lower throughput than DMP-TM. For HTM-SGL, this is due to the fact that stock level and delivery operations, that do not meet HTM's capacity limitations, are much longer than payment operation. This hinders parallelism and thus limits throughput gains and scalability of HTM-SGL. While for HyTinySTM, also in this case the problem is rooted to the high instrumentation costs of the HTM path. NOrec, HyNOrec and HyTL2 incur performance losses due to the fact that payment operation has very high contention. After 32 threads, neither DMP-TM nor DMP-TUNE scale. This happens despite the commit breakdown plot (and the abort rate) incurred by these solutions do not show any significant spike. Further analysis verified, though, that, above 32 threads, the Instruction Per Cycle drop severely, with a corresponding spike in the number of stalled cycles — which suggests that, increasing the thread count, the bottleneck for DMP-TM eventually becomes contention to some physical resource, probably memory or some micro-architectural resource of the processor.

Finally, the right column of Figure 4.6 shows the throughput results for a workload with low degree of partitionability, as reflect by the high system calls to commits ratio. Due to the high number of system calls, DMP-TM incurs 16× lower throughput compared with TinySTM at 80 threads. Again, thanks to its self-tuning ability, DMP-TUNE is able to fall back to TinySTM and achieve similar throughput.

4.6 Related Work

As this work targets HyTM, it has relations with the body of literature that focused on enhancing the efficiency of HTM systems using software mechanisms.

In this context, a first branch of works aimed to enhance the efficiency of HTM systems that use a fallback path a single global lock (SGL). Afek et. al [113] introduced using an auxiliary lock to minimize the avalanche effect, where cascading aborts happens when the pessimistic lock is acquired. Calciu et. al [73] suggested lazy subscription of the global lock to minimize the possible window of conflict between HTM transactions and the fallback path. Diegues and Romano [89] leveraged several on-line tuning mechanisms to decide upon when to acquire the pessimistic fallback lock. SEER [79] used probabilistic techniques to appropriately schedule conflicting transactions and reduce the frequency of activation of the fallback path. POWER8-TM, Chapter 3, minimizes the effect of capacity aborts by using *Rollback-Only Transactions*, which are available on POWER8 processor (see Section 2.2.5 to fit larger transactions in hardware.

DMP-TM has clearly strong relations with the research in the area of HyTM systems, which, like DMP-TM, aim to support the concurrent execution of both HTM and STM transactions. HyNOrec relies on a simple instrumentation mechanism on HTM side (that only requires to increase the sequence lock used by NOrec), which, although being relatively lightweight, exposes HTM transactions to spurious aborts in presence of concurrent commits by (non-conflicting) STM transactions. Reduced HyNOrec extended HyNOrec's design with

a transactional chopping mechanism to further enhance its efficiency. However, both these solutions are limited to employing NOrec, whose design is known to be optimized for low thread count [9].

Conversely, DMP-TM is STM agnostic and thus can be integrated with *orec*-based STM implementations that were shown to achieve much higher scalability levels [7, 70]. Further, DMP-TM does not incur any extra instrumentation on top of its HTM path and thus does not suffer from spurious aborts. Existing HyTM algorithms that support more scalable *orec*-based implementations do exist. Unfortunately, though, they either suffer from spurious aborts (e.g., HyTL2 [26]), analogous to HyNOrec, or require instrumenting the read, write and commit operations of HTM transactions (e.g., Invyswell [27] and Hybrid-LSA [25]), incurring significant overheads — as quantified via the study in Section 4.5.

Ruan et. al proposed Hybrid Cohorts [116], a HyTM that spares both HTM and STM transactions from any extra-instrumentation by only allowing HTM to commit when there are no active STM transactions. Analogously, PhTM [115] aims to reduce synchronization overheads between STM and HTM by executing them in alternate phases. Unlike DMP-TM, both these solutions prohibit concurrency between HTM and STM.

The idea of using memory protection mechanisms in the context of TM was first proposed by Abadi et. al [118] to ensure correct synchronization between a STM system and non-transactional code (i.e., strong atomicity [63]). DMP-TM builds on analogous base building blocks, i.e., OS based memory protection mechanisms, in a different context (HyTM) and to solve a different problem: enable concurrent, yet safe, execution between HTM and STM transactions.

Finally, DMP-TM is related with prior works that investigate the partitionability of workloads not only for TM applications [117], but also in relational [119] and NoSQL [120] domains. Particularly relevant for DMP-TM is the work by Riegel et al. [117], which has first shown that, even in irregular TM applications like the ones included in the STAMP suite, it is often possible to encounter disjoint data partitions that can benefit from the adoption of distinct (software-based) synchronization schemes. It should be noted, though, that DMP-TM considers different synchronization mechanisms than the ones targeted by Riegel et al., and, hence, a different definition of "partitionability". Also, unlike the solution proposed by Riegel at al., DMP-TM is designed to operate (and can achieve significant performance gains) also in presence of workloads that are not perfectly partitionable.

4.7 Summary

This chapter presented DMP-TM, a novel HyTM algorithm that exploits a key idea: leveraging operating system-level memory protection mechanisms to detect conflict between HTM and STM transactions. This innovative design allows for employing highly scalable Orec-based STM implementations, while avoiding any instrumentation on the HTM path. DMP-TM demonstrated robust performance in an extensive evaluation achieving striking gains of up to $\sim$20$\times$ compared to state of the art HyTM systems.

As a concluding remark, it is worth stressing that DMP-TM cannot operate on current

Intel's HTM implementations, which, unlike IBM's, do not provide support for a simple feature: reporting information on the address that caused an access violation from within a transaction. Hopefully, the performance benefits achievable by DMP-TM will motivate other CPU manufacturers, besides IBM, to integrate such features in their future CPU generations.

Chapter 5

Green-CM

Whether TM is implemented in hardware, software or HyTM, its optimistic nature may lead to a waste of work, i.e., a waste of energy. This chapter proposes, Green-CM, a contention manager designed to enhance efficiency of TM systems, in terms of both performance and energy consumption. It combines different back-off policies in order to take advantage of Dynamic Frequency and Voltage Scaling (DVFS) and introduces an energy efficient implementation of the `wait` mechanism. Some passages in this chapter have been quoted verbatim from [44].

5.1 Problem

Most TM implementations take a speculative approach, and run transactions in a lock-free, optimistic fashion. This makes them prone to incur high abort rates in presence of high contention workloads, which can lead to severe degradation of both performance and energy efficiency. It is the responsibility of the Contention Manager (CM) module to reduce the detrimental effects of contention, by deciding which transactions should be aborted in case of a conflict, and when to restart the aborted transaction.

Nowadays, energy efficiency is becoming an increasingly relevant factor for a wide range of systems, from sensors or mobile nodes that run on batteries, to data centers, whose scalability is nowadays constrained by their energy costs [30]. Quite surprisingly, though, despite the literature on CM is quite vast, most CM systems have been designed to maximize solely performance and have largely overlooked the problem of energy efficiency [75–78]. Only a very limited number of works have investigated CM designs aimed at maximizing energy efficiency [121, 122], but none of them has been evaluated in a real system.

5.2 Overview

Green-CM relies heavily on Dynamic Voltage and Frequency Scaling (DVFS) [39] to enhance energy efficiency. DVFS is an architectural feature that is widely employed in modern processors [123, 124] in order to enhance their energy efficiency. DVFS allows various cores

of a processor (and/or various processors in a multi-socket system) to adjust dynamically the voltages and frequencies at which they operate: on the one hand, this allows both for reducing the energy consumed by idle cores; on the other hand, it allows for increasing the frequency of active cores, as long as the number of idle cores is large enough to ensure that the global thermal envelope remains within acceptable margins. To achieve energy efficiency, Green-CM leverages three innovative mechanisms.

Energy efficient backing off. Green-TM relies on a novel, energy efficient implementation of one fundamental building block at the basis of most existing CMs and that can have a strong impact on energy consumption: the `wait` primitive that is used whenever the CM decides to block a conflicting transaction for some period of time. The proposed solution uses a hybrid approach that alternates between two implementations, based, respectively, on spinning and sleeping. The latter allows for effectively reducing energy consumption, when employed on a DVFS enabled processor, but incurs long latencies due to the need for invoking a system call; spinning has opposite advantages and drawbacks: it is accurate also for very short backing off periods, but suffers of high energy costs. By leveraging on both implementations in synergy, Green-CM aims to achieve the best of both worlds, namely low energy consumption and high accuracy. The energy efficient implementation of the back-off mechanism integrated by Green-CM is presented in Section 5.3.1.

Energy-aware contention management policy. Green-CM introduces an innovative, *Asymmetric* CM (ACM) policy, which aims to take advantage of ACM is based on the key idea of promoting the exploitation of DVFS boosting capabilities via the usage of asymmetric back-off policies. More in detail, ACM combines aggressive and conservative (i.e., linearly vs exponentially increasing) back-off policies, in order to promote the dynamic creation, at medium/high contention scenarios, of two sets of threads: 1) threads that are likely to be backing-off, allowing the corresponding processor to enter deep sleep states, and 2) threads that spend most of their time executing transactions, and which can run at higher frequencies thanks to DVFS. The ACM policy is detailed in Section 5.3.2.

Self-tuning. Green-CM makes extensive use of lightweight reinforcement learning techniques to dynamically adapt its internal parameters and specifically: *(i)* determining automatically in which scenarios spin vs timer-interrupts based implementations should be used, and *(ii)* what degree of asymmetry should be used when determining the Contention Management back-off policies. Different variants of gradient descent based controllers were proposed and evaluated to tackle each of these two problems, both individually and in conjunction. Self-tuning mechanism used within Green-CM are detailed in Section 5.3.3.

Finally, Section 5.4 shows the results of an extensive experimental study, based both on complex TM benchmarks (STAMP and STMBench7) and recent TM-based implementations of real-life applications (Memcached). Green-CM was evaluated against 6 alternative CM implementations from the perspective of performance and energy consumption. The experimental data shows that Green-CM achieves average gains of 25% when considering

joint energy-performance metrics, i.e., energy-delay product (EDP), with peak gains that extend up to 2.35× lower EDP. The effectiveness of the proposed self-tuning mechanisms was also assessed, which achieved performance that is on average within 15% from, and sometimes even superior to, the best, manually identified, static solutions.

5.3 Description

The high level architecture of Green-CM (see Figure 5.1) is composed by three main components can be identified: the Asymmetric Contention Manager (ACM), a hybrid implementation of the back-off primitive, and the Controller.

Upon the abort event of a transaction, the first component to be triggered is the Asymmetric Contention Manager. This module is in charge of determining the duration of the back-off phase for the transaction and, as discussed more in detail in Section 5.3.2, it determines whether to use an aggressive (i.e., linear) or a conservative (i.e., exponential) back-off policy depending on two factors: *(i)* the CPU core on which the corresponding thread is running, and *(ii)* the boosting degree (noted $\mathcal{B}$ in the following), i.e. a tunable parameter (dynamically configured by the Controller) that allows for controlling how many threads should use each of the two available back-off policies.

The duration of the back-off phase is next provided as input to a hybrid implementation of the `wait` primitive, which is described in Section 5.3.1. The hybrid `wait` primitive uses either a spin-based or a sleep-based implementation depending on two factors: *(i)* the duration of the back-off phase, and *(ii)* the value of two parameters, noted α and $\mathcal{T}$, which represent, respectively, the number of spin cycles executed in a time unit, and the minimum back-off duration for using a sleep-based approach. Analogously to the case of $\mathcal{B}$, the tuning of α and $\mathcal{T}$ is delegated to the Controller.

Finally, the Controller is in charge of self-tuning $\mathcal{B}, \alpha$ and $\mathcal{T}$. To this end, this module gathers periodic measurements on the throughput and energy consumption over the last time window, and uses this information to implement a lightweight, on-line self-tuning scheme. Controller module shall be discussed in Section 5.3.3.

5.3.1 Hybrid Back-off Implementation

The main purpose of CMs is to reduce the detrimental effects of contention on the efficiency of TM systems. This objective is pursued by reducing the likelihood that threads executing conflicting transactions execute at the same time. One of the most common techniques to perform this is to force threads to back-off for a certain period of time when they encounter contention, before re-starting the aborted transaction. The duration of the `wait` period is determined by the back-off policy employed by the CM (e.g., exponential back-off) and can be expressed either in terms of processor cycles (e.g., number of iterations during which to spin) or in time units (e.g., nanoseconds).

In principle, there exist two ways of implementing the `wait` primitive: *(i)* spinning, i.e., busy-waiting, in an empty loop, or possibly by invoking at each iteration "pipeline-friendly" assembly instructions, such as `pause` in x86 architectures; *(ii)* sleeping by invoking the

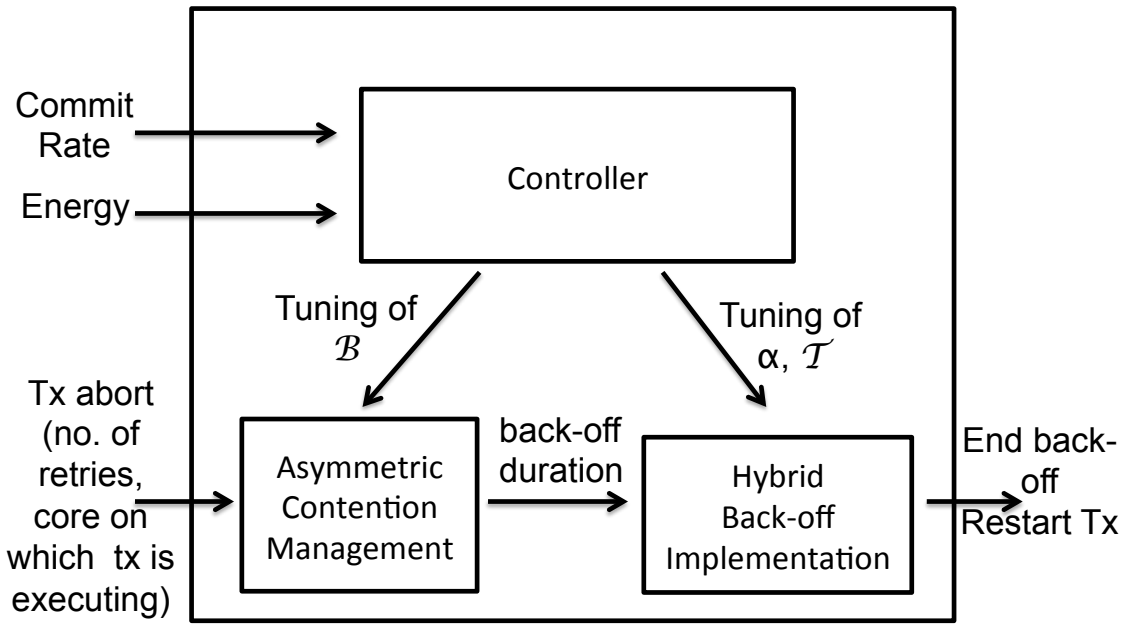

Figure 5.1: Architecture of Green-CM.

`sleep` system call. The two methods exhibit clear trade-offs for what concerns performance and energy consumption. Busy waiting has very fine granularity, but, from the perspective of energy consumption, it is strongly inefficient. Sleeping, on the other hand, achieves low energy consumption but provides coarse granularity (various tens of micro-seconds in recent architectures/OSs [125])), which can have a detrimental impact on the effectiveness of the CM policy.

Since existing CMs have focused on optimizing performance, basically neglecting the issue of energy efficiency, existing implementations rely solely on spin-based approaches — which, do not suffer of the accuracy issues of sleep-based approaches for short back-off periods. The drawback is that they miss the opportunity of reducing energy consumption when back-off times are sufficiently large to be effectively supported using sleep-based implementations.

The hybrid approach that proposed in this chapter, and whose pseudo-code is reported in Algorithm 8, is based on a simple, yet effective idea: using a sleep-based or a spin-based implementation depending on the duration of the back-off period. The intuition is that spin-based implementations are ideal for "sufficiently short" back-off periods, whereas sleep-based ones work best for "sufficiently long" back-off period.

Despite the idea may at first glance appear relatively straightforward, it does hide two non-trivial, and closely intertwined, issues:

1. Spin-based and sleep-based implementations operate using different time scales: the latter expresses the back-off duration in real-time units (e.g., nano-seconds), whereas

Algorithm 8 Green-CM: hybrid back-off mechanism

```
1: function WAIT(waitCycles)                                   ▷ Back-off ...
2:     if waitCycles/α ≤ T then                                ▷ for a "short" period,
3:         while waitCycles ≠ 0 do                             ▷ so just spin meanwhile
4:             waitCycles− −
5:     else                                                    ▷ for a "long" period,
6:         sleep(waitCycles/α - min_sleep)                     ▷ so sleep instead.
7: end function
```

the former uses spin cycles, or, equivalently, processor cycles. In order to hide both implementations under the same interface, and use the two transparently, it is necessary to reconcile their time scales, by identifying a conversion factor, note as α, in order to map processor cycles to real-time (or vice-versa).

The issue here is that, in modern processors, the number of spin (or processor) cycles executed within a time unit can vary significantly depending on the impact that the workload's characteristics have on architectural aspects like DVFS, pipeline and caching.

2. What is the minimum value of the back-off duration, which denoted by $\mathcal{T}$, for which sleep-based implementations are more efficient (from a joint energy-performance perspective) than spin based ones? Ideally, the value of $\mathcal{T}$ should be set to the minimum back-off duration for which the gains in terms of energy consumption achieved by sleeping outweigh the performance losses due to its relatively lower accuracy.

One additional noteworthy aspect is that, in order to enhance the accuracy of the sleep-based implementation, in Line 6, the requested sleep duration is adjusted before invoking the sleep system call. More in detail, the minimum latency for executing a sleep system call (i.e., for executing `sleep(0)`), which is noted as *min_sleep* in the pseudo-code, is subtracted from the target back-off duration (i.e., $\frac{\text{waitCycles}}{\alpha}$). In fact, whenever the sleep system call is called with x as input parameter, the actual latency for the execution of sleep is equal to $min\_sleep+x+err$, where err is an error factor depending on the actual sleep accuracy (e.g., hardware timer resolution). This latency can be easily measured experimentally, and by taking it into account, one can significantly enhance accuracy when the back-off duration is of the same order of *min_sleep*.

As it will be discussed more in detail in Section 5.3.3, the identification of the correct value of the parameters α and $\mathcal{T}$ plays a crucial role in determining the efficiency of the CM scheme. Green-CM addresses this problem via a light-weight, on-line self-tuning mechanism, which is also detailed in Section 5.3.3.

5.3.2 Asymmetric Contention Management

Ideally, a CM may take advantage of the DVFS capabilities provided by modern CPUs by scaling down the frequency of a core whenever a transaction has to be aborted and backed-off, and scaling up the frequency of that as soon as the back-off period completes.

If a sufficient number of transactions are in the back-off state, the CM could even request to boost the frequency of some cores, provided that the thermal envelope of the corresponding CPU is within the safety margin.

Unfortunately, controlling the dynamic frequency scaling mechanism requires issuing system calls, which induce prohibitive costs [125] and would largely outweigh the gains achievable via DVFS.

The idea at the basis of ACM is to approximate such an ideal, yet impractical CM policy, by using a lightweight design that aims at favoring the spontaneous activation of hardware-controlled DVFS mechanisms. Modern CPUs, in fact, identify in an automatic fashion the opportunity to boost the frequency of subset of cores, whenever a sufficient number of cores in the same CPU have entered a sleep state (and are hence executing below the nominal frequency).

In order to make the contention management scheme DVFS-aware, Green-CM exploits a simple idea, which is: adopting an asymmetric approach that divides threads into two categories: *(i) boosted* threads that are active on cores to be boosted, and *(ii)* threads executing on cores to be pushed towards lower operating frequencies. This can be achieved by letting the CM treat these two categories in an asymmetric fashion; the threads to be boosted will be backed off for linearly increasing periods, while the other category will be backed off for exponentially increasing periods.

Under such arrangement, and considering a hybrid implementation of the back-off mechanism, like the one described in the previous section, boosted threads are likely to be either executing transactions or spinning, as they will most likely back-off for short periods. The other threads, on the other hand, will tend to back-off for longer periods. Hence, they are more likely to use the sleep-based back-off implementation and to have their cores enter deeper sleep states.

Note that in order to favor the activation of the hardware-controlled DVFS mechanism, the selection of which threads should be boosted has to be made in an architecture-aware fashion. In fact, in order to create the preconditions for DVFS to accelerate the frequency of the core on which a boosted thread is executing, the number of boosted threads active in each CPU should not exceed the maximum number of cores $\mathcal{M}$ that can simultaneously execute at frequencies higher than the nominal ones. For instance, in AMD Opteron CPUs such as the ones used in our experimental evaluation, at most two cores out of the 8 cores available in each processor can enter the boosted state, when the other 6 are sleeping. This architecture-dependent parameter is taken into account by the ACM, which scatters the $\mathcal{B}$ boosted threads across the available CPUs, assigning at most $\mathcal{M}$ boosted threads per CPU.

The decision of how many boosted threads to use, or boosting degree ($\mathcal{B}$), which will be further discussed in Section 5.3.3 is non-trivial, as the optimal tuning is in general workload-dependent. As already mentioned, the task of automating the tuning of $\mathcal{B}$ is delegated to the Controller module

5.3.3 Controller

As already mentioned, the Controller relies on on-line self-tuning techniques in order to identifying the values of the parameters $\alpha, \mathcal{T}$ and $\mathcal{B}$ that yield maximum energy-efficiency. The design of the self-tuning will be discussed in Section 5.3.3. After, though, an experimental data aimed at highlighting the relevance of tuning each of these three parameters will be presented. The analysis of this data will also allow us to obtain some important insights that will drive the design of the self-tuning mechanisms employed by the Controller.

The need for self-tuning

The results reported in this section, and in the remainder of the chapter, were obtained running with 64 threads on a machine equipped with an AMD Opteron 6272 CPU running Linux 3.13 and equipped with 32 GB of RAM.

First, a sensitivity study is performed to evaluate the tuning of α and $\mathcal{T}$. To this end, two benchmarks of the STAMP benchmark suite, which generate workloads with distinct characteristics are considered: *intruder* generates relatively long transactions that have a high contention probability; transactions in *kmeans*, conversely, are relatively short and less prone to aborts.

In Figure 5.2, the number of active threads is set to 64. The EDP obtained when varying α from 100 to 10^7 with $\mathcal{T} = min\_sleep$ (the minimum sleep granularity) is reported after being normalized with respect to the EDP obtained when using the optimal values for α and $\mathcal{T}$, identified via an exhaustive off-line search. The rationale for setting the threshold $\mathcal{T} = min\_sleep$ is that this represents the minimum value for which the input argument of the sleep system call, in line 6 of Algorithm 8, can be guaranteed to be positive, and hence meaningful.

The plot allows to draw two interesting conclusions. On the one hand, the optimal value of α for the two benchmarks is significantly different, being equal to 5K for *intruder* and to 250K for *kmeans* — a difference of two orders of magnitude. Also, if one uses the optimal setting of α for *kmeans*, resp. *intruder*, with Intruder, resp. *kmeans*, the EDP is more than 2×, resp. 5× higher. These data *intruder* highlight the relevance of appropriately tuning this parameter.

On the other hand, by setting statically $\mathcal{T} = min\_sleep$ (and properly tuning α) the EDP obtained is very close to (i.e., at most 5% higher than) the EDP obtained by using any alternative value of $\mathcal{T}$. This is true despite the fact that the two considered benchmarks have radically different workload characteristics. In fact, it was experimentally verified across the entire set of benchmarks considered in this chapter (whose full list is provided in Section 5.4.1) that the quality of the solution obtained by using this simple heuristic is always very close to the optimal one. In the light of these considerations, and in order to maximize the convergence speed of the self-tuning mechanisms employed by the Controller, the dimensionality of the optimization problem is reduced by setting $\mathcal{T} = min\_sleep$.

Let us now analyze the effects of using different values for the $\mathcal{B}$ parameter. To this end 5 popular TM benchmarks are considered, namely *intruder*, *kmeans* from the STAMP

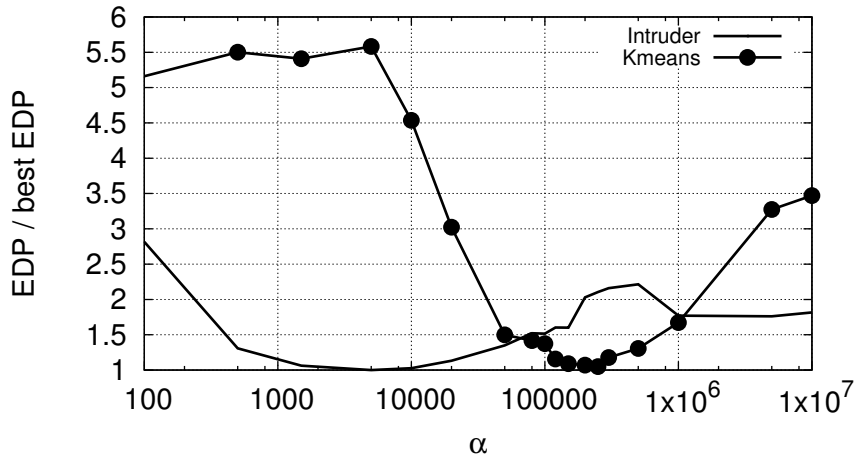

Figure 5.2: EDP of different static configurations of α with $\mathcal{T} = min\_sleep$, normalized with respect to the EDP of the best, off-line identified, configuration for α and $\mathcal{T}$.

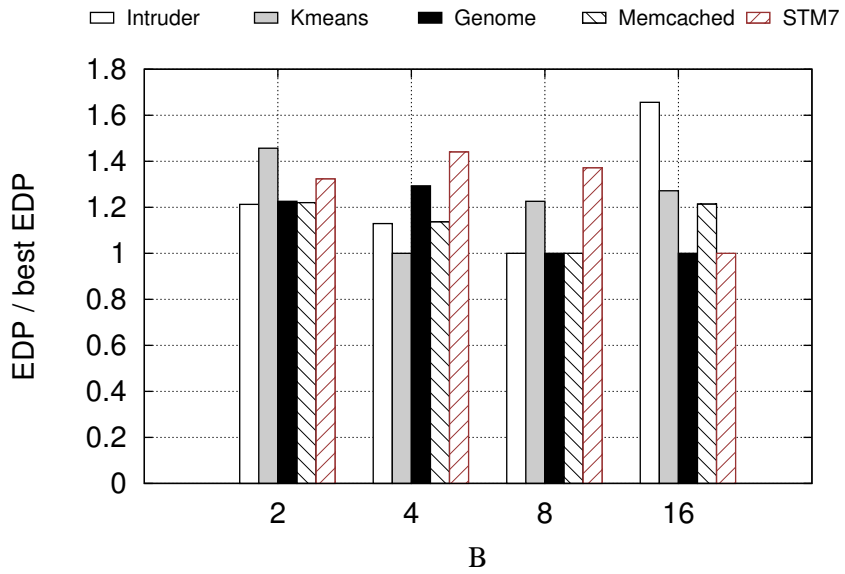

Figure 5.3: EDP for different static values of $\mathcal{B}$ normalized to the EDP obtained using the best setting of $\mathcal{B}$.

suite, Memcached and STMBench7, which are run using 64 threads in total. $\mathcal{B}$ is treated as the independent parameter, while setting α and $\mathcal{T}$ to their optimal, off-line determined, values. In the machine used in our study, at most 16 cores can operate above the nominal frequencies (when the remaining 48 are in sleep state). Accordingly, the maximum value of $\mathcal{B}$ is set to 16. In Figure 5.3 the EDP obtained using different static values of $\mathcal{B}$ is reported after being normalized to the EDP obtained using the best setting of $\mathcal{B}$. From the plot, it is obvious that the optimal setting of $\mathcal{B}$ varies significantly with the considered benchmark: for *intruder*, for instance, EDP degrades by around 70% if $\mathcal{B} = 16$, since the contention level generated by having so many threads using an aggressive, linear back-off policy grows unacceptably large; the opposite is true for STMBench7, for which the optimal $\mathcal{B}$'s value is 16, and using lower values can yield up to 40% increase of EDP.

Design of the self-tuning scheme

As discussed in Section 5.3.3, setting $\mathcal{T} = min\_sleep$ reduces the dimensionality of the on-line optimization problem that the controller has to tackle, which is reduced to identify the optimal tuning of α and $\mathcal{B}$.

The Controller tackles this problem by employing a lightweight, model-free on-line search approach [91], which identifies the optimal values of the target parameters by exploring alternative points in the $\alpha \times \mathcal{B}$ space. There are two main design decisions that are the basis of any model-free on-line optimization algorithm: *(i)* how to explore the search space and *(ii)* when to stop exploring and start exploiting the available knowledge.

In the design of the Controller, alternative policies for tackling each of these two problems were considered. Each of them is described in the following, while their evaluation is postponed to Section 5.4.1.

How to explore the search space. The exploration policies represent variants of the classic hill-climbing algorithm, which operates as follows: at each iteration the neighbors of the current configuration are tested and the one that maximizes the target metric is set as the new configuration for the next iteration. The basic hill-climbing algorithm suffers of three main problems, which can be addressed by considering several additional mechanisms, described in the following:

Local minima. Due to the localized nature of its search policy, hill-climbing is well known to be prone to get stuck in local minima. A commonly employed solution to this problem is to force random jumps with a fixed, small probability. This variant is noted *jmpX*, where X is the jump probability.

Curse of dimensionality. The number of neighbors for a configuration grows exponentially with the dimensionality of the search space [91]. In order to circumvent this issue, two alternative exploration policies are considered: *(i)* treat the two dimensions α and $\mathcal{B}$ as tunable in a completely independent fashion, and run two hill-climbing based optimizers, each targeting a different dimension, in parallel and without any synchronization. This

policy is noted as *independent*. *(ii)* Subdivide the exploration in phases, and during each phase optimize along exclusively one dimension, changing the target dimension whenever a phase ends. This policy is noted as *alternate*.

Slow convergence in large domains. The hill-climbing can converge after an unacceptably high number of explorations if the domain that is being explored spans a broad range of values and the granularity used to identify the neighbors of the current configuration (also called, exploration step) is too small. On the other hand, using overly large exploration steps increases convergence speed, but can have a detrimental effect on the quality of the identified solution. In the problem at hand, as already noted, α spans a very broad domain (from a few hundreds to about one million). To cope with this issue, when moving along the α dimension the controller uses an adaptive exploration step: it starts by adopting a large (125K) exploration step, which it halves whenever the direction of exploration along the α dimension is inverted (because a sub-optimal value is found) till a minimum value for the exploration step (1K) is reached. A fixed exploration step equal to one is when moving along the $\mathcal{T}$ dimension.

Exploring vs exploiting. The hill-climbing approach never stops exploring, meaning that even when it identifies a minima, it keeps on oscillating around it forever. This has the advantage of making it prone to react to changes in the function being optimized (e.g., imputable for instance to shifts of the application's workload). On the down side, if the function is stable, moving away from the (local) optimum, and re-exploring a configuration that is known to be sub-optimal, means incurring a certain penalty. A simple heuristic that can be used to tackle this problem is to detect subsequent oscillations around the current local minimum, and stop explorations, which is called `stabilizing`.

5.4 Evaluation

This section aims to quantitatively evaluate Green-CM from a twofold perspective. It starts by assessing the effectiveness of the various self-tuning approaches considered in Section 5.3.3. Then, evaluating the performance, energy consumption, and EDP of Green-CM with respect to state of art CM solutions.

As already mentioned, Green-CM was designed to work with both hardware and software based TM implementations. In this study, TinySTM (see Section 2.2.4) is selected as reference TM implementation, as it has been shown to excel in a wide range of workloads [70]. It also comes with different contention managers including exponential back-off with busy waiting. A set of 4 well-known TM benchmarks is considered: the *intruder* and *kmeans* from the STAMP benchmark suite and Memcached and STMBench7 (see Section 2.4), both generating 50% read and 50% write transactions. All reported results are the average of at least 5 runs.

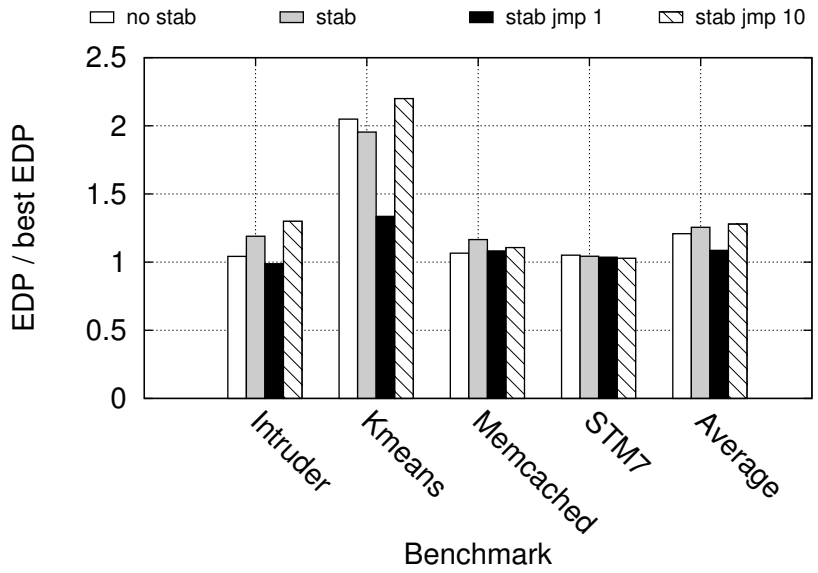

Figure 5.4: Normalized EDP across different benchmarks using different strategies for self-tuning individually α.

5.4.1 Tuning Strategies

To start, the effectiveness of the self-tuning algorithms when operating on each dimension of the search space $\alpha \times \mathcal{B}$ are evaluated individually. This preliminary analysis will allow to circumscribe the combinations of self-tuning algorithms that will be evaluated to optimize α and $\mathcal{B}$ in conjunction.

Individual tuning of α and $\mathcal{B}$.

For the individual optimization of α and $\mathcal{B}$, the number of active threads is fixed at 64 and consider four alternative self-tuning strategies: *(i) nostab*, a non-stabilizing policy that does not perform probabilistic jumps; *(ii) stab*, a stabilizing policy that does not perform probabilistic jumps; *(iii) stab jmp1* and *(iv) stab jmp10*, a stabilizing policy that perform random jumps with probability 1% and 10%. When evaluating the self-tuning of α, $\mathcal{B}$ is set to 0. When self-tuning $\mathcal{B}$, α is set to the corresponding optimal, off-line found value for $\mathcal{B}$.

Figure 5.4 shows the EDP across different benchmarks when using the different tuning strategies for α, normalized to the EDP obtained when using the (per-benchmark) optimal, off-line found value of α. By looking at the plot, it can be deduced that the stabilizing tuner then performing random jumps with 1% probability outperforms all others approaches, achieving an EDP that is only 8% larger than the optimal static solution (identified after an exhaustive off-line exploration). The reason behind this is the fact that stabilization minimizes the cost paid oscillating around a minimum. Also, a small jump probability is

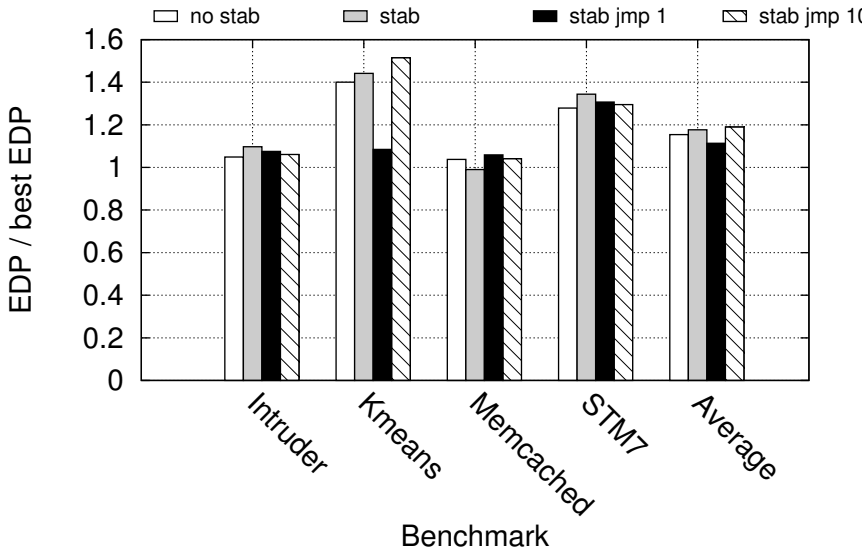

Figure 5.5: Normalized EDP across different benchmarks using different strategies for self-tuning individually $\mathcal{B}$.

sufficient to allow the tuner to escape from a local minima, without excessively hindering performance with overly frequent random explorations.

Analogous considerations can be drawn by analyzing Figure 5.5, which shows the results for the same study conducted on the tuner of $\mathcal{B}$. The considered strategies behave similarly to the previous case, although the relative differences between them are smaller. It can be argued that this is a consequence of the fact that identifying the optimal tuning of $\mathcal{B}$ is a relatively easier problem, given that the corresponding domain is much smaller than the one of α.

Joint tuning of α and $\mathcal{B}$.

Next, different strategies for tuning α and $\mathcal{B}$ in conjunction are considered: *(i) independent stab jmp1*, two independent tuners, using stabilization and random jumps with 1% probability. *(ii) bidim stab jmp1*, same as above, except that a single learner is used that explores all the current neighbors in the bi-dimensional $\alpha \times \mathcal{B}$ space; *(iii)* `stab jmp X - stab`, an alternate policy, which starts by exploring the α space until it stabilizes. In phase 2, an exploration in the $\mathcal{B}$ space is performed till stabilization. In phase 3, and in the subsequent odd phases, it optimizes α performing random jumps with probability X% until it stabilizes on a new optimum configuration. In phase 4, and in the subsequent even phases, it explores the $\mathcal{B}$ dimension until it stabilizes.

Figure 5.6 shows the EDP normalized the best static configuration which was found by testing all the different combinations of both $\mathcal{T}$ and $\mathcal{B}$ offline. An interesting fact that

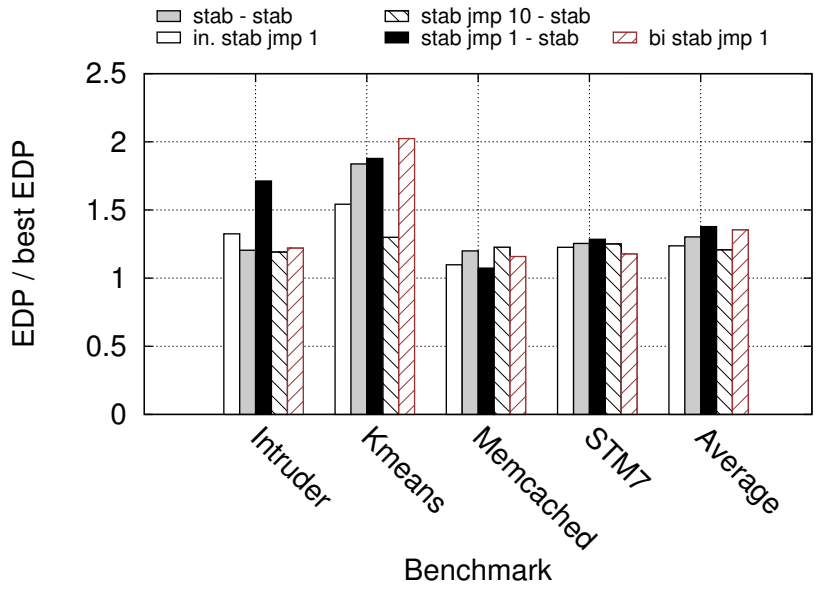

Figure 5.6: Normalized EDP across different benchmarks using different strategies for coupling the two tuners.

can be deduced from these results is that using higher probability of random jumps after performing a stabilization yielded better results as compared to using a single learner. It can be argued that this can depend on the fact that, in order to escape from a local minimum, a larger number of attempts is required, on average, in a bi-dimensional space. Hence, using a larger jump probability is more beneficial in this scenario. The 2nd best option, with an only marginally higher EDP value with respect to stab jmp 10 - stab is represented by the independent tuners. This suggests that, despite the lack of synchronization between the two tuners, they can still successfully crawl the search space and quickly identify high quality solutions.

Evaluating Green-CM

In this section Green-CM is compared, using the stab jmp 10 - stab tuner, with respect to the following state of the art CMs (see Section 2.2.7): suicide, karma, timestamp (ts), aggressive (agg), exponential back-off with sleep for `wait` implementation (sleep) and exponential back-off with spin for `wait` implementation (spin).

Figure 5.7, shows the EDP, energy consumption and running time for *intruder* and *kmeans*. Figure 5.8 shows the EDP, energy consumption and commit rate for STMbench7 and Memcached. The performances of the considered CMs is normalized with respect to the ones of Green-CM, defining the normalization in such a way to guarantee that values higher (resp. lower) than one mean worse (resp. better) performance than Green-CM, independently of the considered metric. This was performed to enhance data visualization,

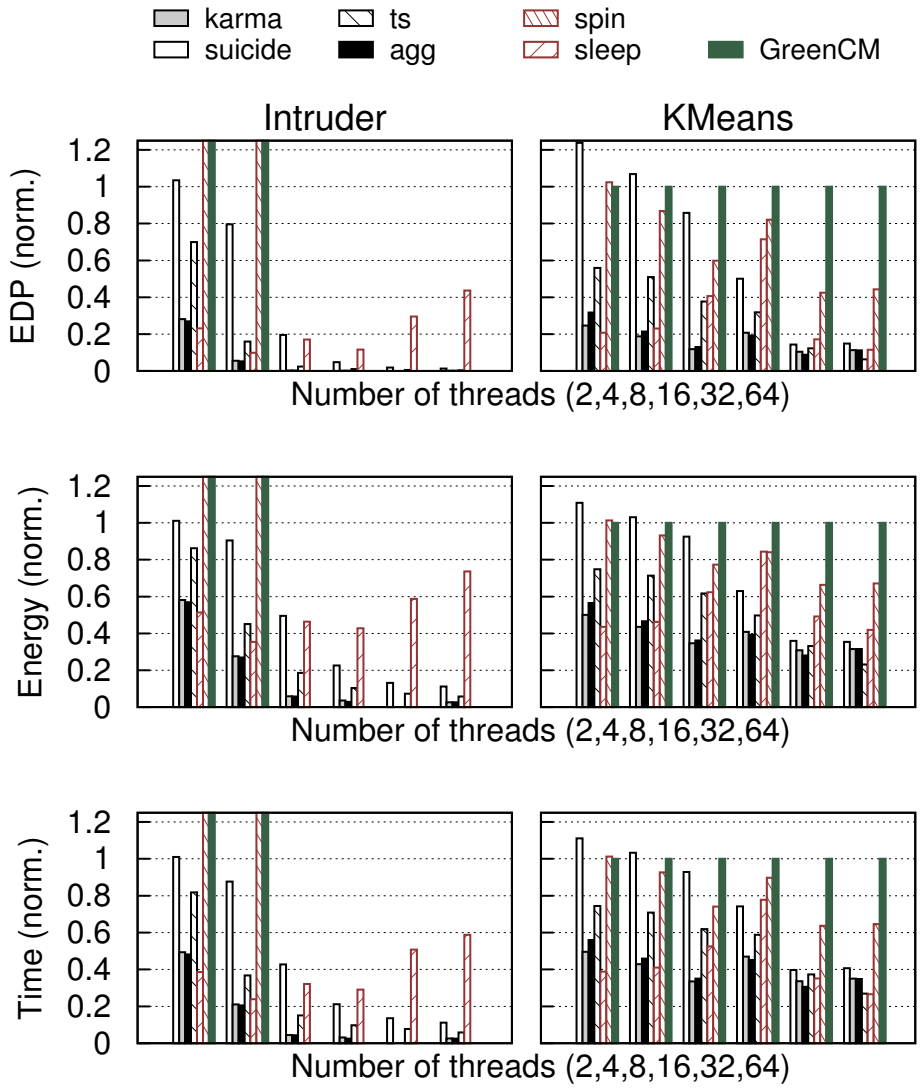

Figure 5.7: EDP, energy consumption and execution time of *intruder* (left) and *kmeans* (right), normalized with respect to Green-CM (Higher is better).

as in some cases Green-CM outperforms existing CMs by various high orders of magnitude.

Overall, Green-CM can achieve up to 2.35 × lower EDP than the best other contention manager with an average of 65% improvement across all benchmarks at 64 threads and 83% improvements at 48 threads with an overall average gain of 25% across all benchmark and thread configurations. Green-CM also achieves better efficiency in terms of EDP in most thread configurations higher than 4 threads for all benchmarks except *kmeans* and Memcached where it is as good as the best competitor up to 8 threads.

It can also be noted that the gains from using Green-CM are more prevalent at higher

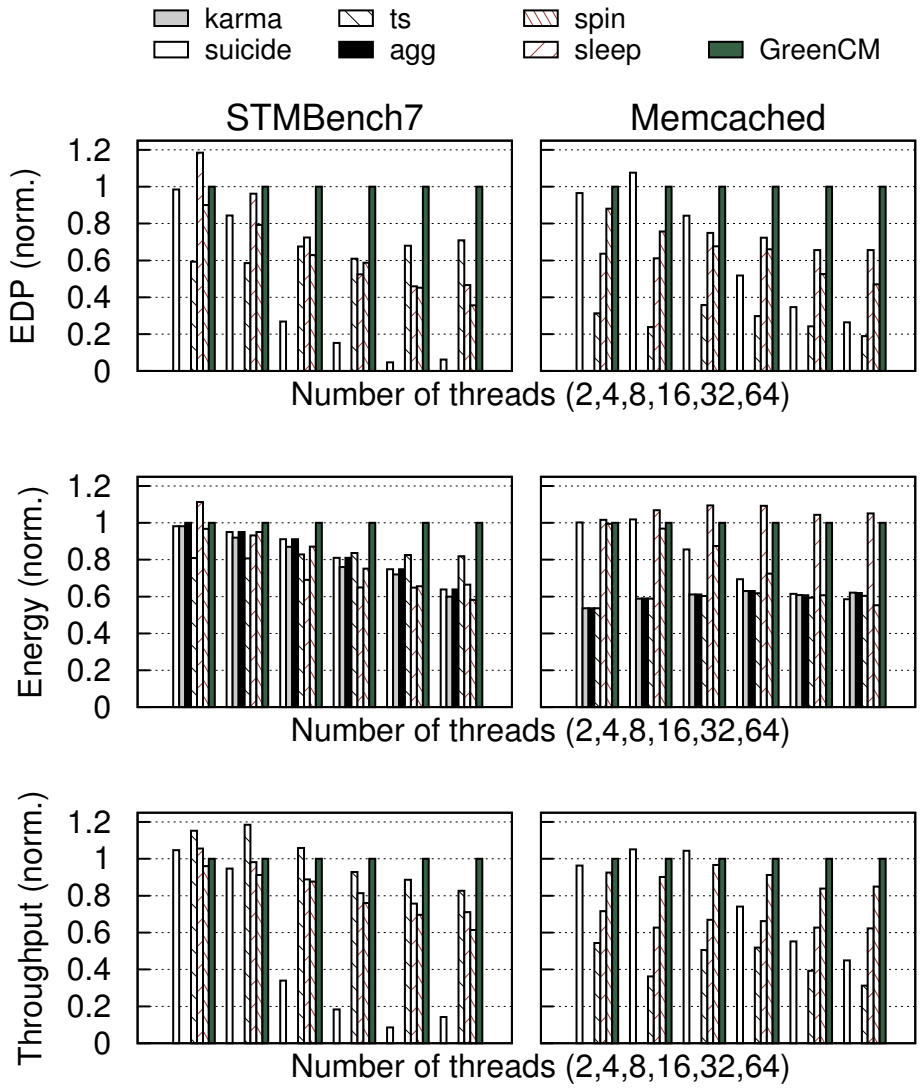

Figure 5.8: EDP, energy consumption and commit rate of STMbench7 (left) and Memcached (right), normalized with respect to Green-CM (Higher is better).

number of threads. This is expected, since the higher the thread count, the higher the contention level, the higher the relevance of a CM.

Finally, to demonstrate the individual impact of using ACM, Green-CM is evaluated with and without asymmetry enabled. Figure 5.9 shows the EDP, energy consumed and execution time for running *intruder* with three different configurations: exponential back-off using a spin-based implementation (spin), tuning only α with $\mathcal{B} = 0$ (no-asym), and Green-CM with both tuners enabled (asym).

From the results, ACM yielded extra gains in terms of both energy and performance

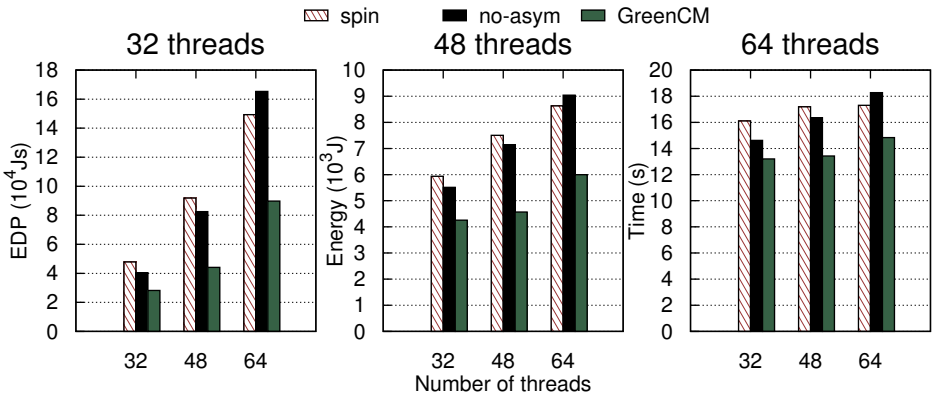

Figure 5.9: EDP, energy consumption and time for *intruder* with and without ACM enabled.

reaching around 25% at 64 threads. These gains can be explained by correlating the results with the average core frequency operating distribution charts shown in Figure 5.10. These charts show the distribution of cores according to their average operating frequency throughout the running time of the benchmarks. Note that the considered AMD processor support 7 different frequency levels, P0, .., P6, where P0 is the highest frequency (3.0GHz), P6 is the lowest (1.4 GHz) and P2 is the nominal frequency (2.1GHz).

It can be seen that between 10 to 25% of the cores reach the maximum boosted state (P0) when asymmetry is enabled, providing evidence on the effectiveness of ACM to favor the spontaneous activation of hardware-controlled DVFS mechanisms. Another aspect that can be noted is that, as the number of threads increase, more cores get to operate at lower frequencies: this is a consequence of the increase of contention, which leads threads to back-off for longer periods. This explains the gains in terms in of energy efficiency compared to exponential back-off using spin for the back-off implementation.

5.5 Related work

Most of the literature of TM is concerned with optimizing TM performance [7–9], but the issue of energy efficiency is much less explored. Indeed, the few existing works on TM that tried to optimize both energy and performance were mainly revolving around energy efficient hardware implementations of TM [126–128].

Sanyal et al. [121] tried to achieve energy efficiency in HTM by clock gating processors upon abort of a transaction. Baldassin et al. [122] adopted a similar idea, although implemented at the software level and integrated with the CM module: using DVFS to lower the frequency of cores upon abort and during the (exponential) back-off phase. Their study was limited to 8 threads only using a simulator that has a very low cost for entering a lower frequency mode, which make this solution largely sub-optimal in practice.

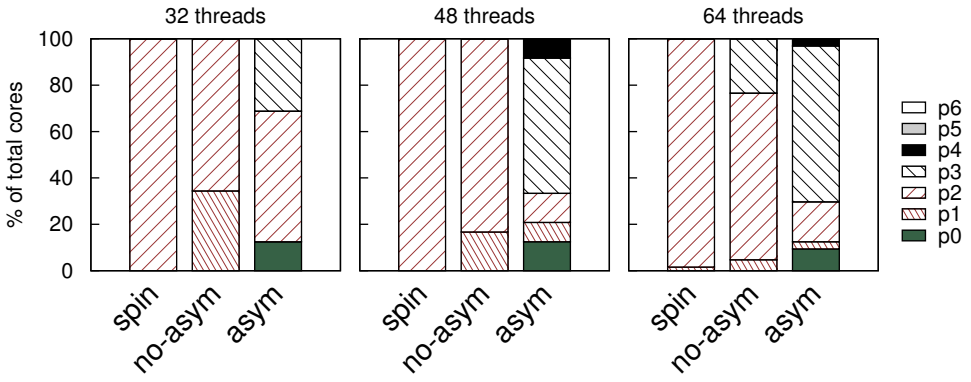

Figure 5.10: Frequency distribution of different thread configurations for *intruder*.

Two studies were performed on energy consumption of TM. Rughetti [87] studied the performance and energy trade-offs of various algorithms; the study showed the necessity of adaptability within TM to minimize data contention as it is the main source of energy consumption. Diegues et al. [70] evaluated the performance and energy efficiency of a large number of TM implementations, including real-life hardware-based (HTM), software-based (STM) and hybrid (HyTM) across a wide range of popular benchmarks. The results of the study highlight that the choice of the right TM implementation is strongly workload-dependent. The Green-CM algorithm has been designed to operate both with STM and HTM, as it does requires no information on the set of items accessed by aborting transactions — an information not available when using HTM [79].

An extensive study of DVFS in latest processors by Intel and AMD was performed in [125]. This work characterized the behavior of frequency scaling on both architectures describing how it can be utilized either automatically (i.e. hardware triggered) or manually (software enabled). This work showed also how to exploit DVFS in order to enhance the performance of a STM called Fast-lane [129], which shows performance improvements at low thread counts. This STM has a master thread whose transactions never abort, by boosting the operating frequency of this thread at the cost of running other threads at lower frequencies. They showed improved performance for some micro-benchmarks.

A large body of research has been aimed to devise algorithms for contention managers [75–78] ranging from very simple policies, such as aggressive CM in which the victim transaction is always aborted, to more complex algorithms that use different heuristics for determining the back-off time upon aborts (e.g., linear vs exponential), or that take into account the amount of work done by the contending transactions (e.g., Karma) and/or when they had been activated. None of these CM policies were evaluated from the perspective of energy efficiency.

Finally, Green-CM is related to the studies that have analyzed the energy efficiency of alternative implementations of locking schemes (whereas Green-CM focuses on TM). For

instance, it was shown [130], that a hybrid combination of busy-waiting and sleeping is the optimal solution in terms of energy-delay product to implement a mutex semaphore. This work, however, leaves unsolved, the issue of determining when to use spinning or sleeping. In Green-CM a similar concept is used in the implementation of the back-off: opting for either spinning or sleeping depending on the duration of the requested backing off phase. There are however at least two fundamental differences. Our technique is employed for a different primitive, i.e. a `wait` and not a mutex, for which the duration of the backing off time period is a priori known, hence it can be exploited to make an informed decision on whether to use spin-based or sleep-based implementations. Further, Green-CM solves the problem of fully automating the decision of when to use spinning or sleeping by means of a lightweight, on-line self-tuning technique.

5.6 Summary

This chapter presented Green-CM, an energy efficient contention manager for TM systems. The energy efficiency of alternative back-off implementations was evaluated using realistic workloads deployed on state of the art STM systems. This study motivated the proposal of an adaptive approach, which determines the most efficient back-off implementation to use, on the basis of the specified back-off period. This building block was then integrated into a novel asymmetric CM policy, which aims to favor the activation of frequency boosting via DVFS mechanisms. The proposed solutions has been shown to achieve significant EDP gains (by up to 2.35×) when compared to state of the art CM policies.

Chapter 6

Final Remarks

Transactional Memory (TM) borrowed the concept of transactions from the database domain and applied it to parallel programming to provide an alternative synchronization paradigm. The TM abstraction allows an easy to use interface; programmers are only required to identify the code blocks that should execute atomically, delegating to the implementation of the TM system the task of ensuring their correct synchronization in presence of concurrent executions. By relieving programmers from the burden of having to reason on how to ensure a correct, yet efficient, synchronization of their applications, TM allows application developers to focus on the logic of their programs, hence increasing their productivity. Along with ease of use, TM also promises performance comparable with complex synchronization mechanisms, such as based on fined-grained locks or on lock-free algorithms.

TM comes in different flavors: it can be implemented as a software library, a hardware feature or a hybrid mix of both. The past 20 years of TM research have shown that there exists no implementation that excels in all workloads, as each implementation has its own merits and limitations which make it a better fit for different applications. Software TM (STM) provides robustness and flexibility but pay high instrumentation cost to track shared data accesses. Hardware implementations of TM (HTM) are very efficient in detecting conflicts but suffer from intrinsic limitations, which are not ought to change in near future due to high costs of hardware modifications, that render them unpractical in many workloads. Hybrid TM (HyTM) systems seek to obtain the best of STM and HTM, by allowing HTM transactions to use some STM implementation on their fallback path.

This thesis identified and tackled three relevant limitations that affect the efficiency of state of the art TM systems, without compromising what is generally regarded as the key feature of the TM abstraction: its ease of use.

Capacity limitations of HTM. The availability of a constrained capacity for storing transactional meta-data is, arguably, one of the major limitations of HTM: transactions that access locations more than what the hardware can accommodate are doomed to fail to commit successfully, even in absence of contention.

The first contribution of this dissertation, POWER8-TM (P8TM), is a technique that

allows for expanding the effective capacity available for HTM applications. P8TM adopts a hardware-aware software design approach, which exploits two hardware features available in the IBM POWER8 HTM: (i) Rollback-Only Transactions and (ii) Suspend/Resume (S/R). P8TM leverages these architectural features via novel software-based synchronization algorithms, which relieve read-only transactions from any capacity limitation, while expanding the effective capacity of update transactions by an order of magnitude achieving up to ∼7× throughput speedups.

Synchronization overheads in HyTM. Despite the number of proposals in the HyTM area, state of the art HyTM implementations still suffer from large synchronization overheads to ensure correctness when HTM and STM run concurrently. Indeed, existing HyTM systems either impose expensive instrumentation on the HTM path, or achieve limited concurrency between transactions executing in hardware and software.

The second contribution of this thesis, DMP-TM, is a novel HyTM system that tackles this issue by exploiting a key novel idea: leveraging operating system-level memory protection mechanisms to detect conflicts between HTM and STM transactions. This innovative design allows DMP-TM to execute HTM transactions without any extra instrumentation, while enabling concurrency with highly scalable STM implementations. DMP-TM demonstrated robust performance in an extensive evaluation achieving gains of up to ∼20× compared to state of the art HyTM systems.

Energy efficient contention management. TM implementations tend to adopt speculative approaches, in which transactions are executed optimistically and aborted in case contention is detected. Such a design enables high scalability in uncontended workloads, but is known to lead to severe performance degradation in presence of high degrees of contention. In order to cope with this issue, TM systems typically rely on Contention Manager (CM) modules, which aim to minimize the detrimental effects of contention by determining which transactions to abort and when to restart them. Despite the abundant research on CM, state of the art CM schemes overlook the problem of energy efficiency — a factor that is increasingly relevant nowadays.

The third contribution of this dissertation fill this gap by introducing Green-CM, the first contention manager explicitly designed to jointly optimize both performance and energy consumption. Green-CM employs an adaptive implementation of the primitive used to back-off threads when conflict occurs, which dynamically adopts either a spin-based or a sleep-based policy based on their actual (workload-dependent) run-time efficiency. Moreover, Green-CM introduces a novel contention management policy, which aims to leverage automatic boosting of core frequencies, available via Dynamic Voltage and Frequency Scaling capabilities of modern processors. Thanks to the synergistic use of these two mechanisms, Green-CM is able to reduce energy-delay-product by more than 2×, when compared to various CM solutions.

A key common trait of the systems presented in this dissertation is that they all incorporate self-tuning mechanisms that ensure robustness even when faced with non-favorable workloads. The self-tuning mechanisms integrated in the proposed solutions rely on lightweight reinforcement techniques that require neither prior knowledge of the target

platform/workload, nor offline learning.

6.1 Future work

The works presented in this dissertation have not only addressed key limitations of state of the art TM systems. They have also opened a number of intriguing research questions that would be interesting to explore in the future.

The first research question raised by this dissertation stems from the observation that two of the techniques proposed in this thesis, namely P8TM and DMP-TM, leverage hardware features that are only available on (some) IBM processors, but are not supported by the manufacturers of other CPUs with HTM support (in particular by Intel's processors). On the one hand, one may argue that the remarkable efficiency benefits provided by P8TM and DMP-TM might motivate, in future, the ubiquitous adoption of the specific hardware features required by these solutions. On the other hand, it is well known that introducing hardware modifications to today's processors is a very complex, expensive and slow process. This consideration motivates therefore future research aimed at investigating the possibility to adapt the software logic of P8TM and DMP-TM in order to achieve interoperability with a broader range of commodity processors.

In the light of the above considerations, a first research question is whether it would be possible to adapt the design of P8TM in order to lift its current dependence on the hardware ability to suspend and resume the execution of transactions. Indeed, this is a hardware feature that is fundamental to enable P8TM's ability to execute read-only transactions in an uninstrumented fashion. In particular, the ability to suspend and resume hardware transactions is exploited by P8TM to allow update transactions to execute a quiescence phase, during which they can monitor the state of concurrent active read-only transactions using non-transactional reads — thus avoiding spurious aborts that would otherwise arise. Thus, in order to enhance P8TM's portability it would be necessary to revisit its current quiescence mechanism, and use alternative mechanisms to let update transactions determine whether it is safe for them to commit.

A possible approach to cope with this problem could consist in forcing update transactions to abort deterministically whenever they find an active read-only transaction (rather than waiting for the latter to complete execution, as in P8TM). Such an approach would preserve safety, but may also severely compromise the efficiency of update transactions, which would become prone to suffer from frequent spurious aborts, especially in read-dominated workloads, even in absence of contention. This is a problem that may be mitigated using scheduling techniques aimed at minimizing the odds for an update transaction to encounter active read-only transactions when trying to commit. Clearly, the vast literature on transaction scheduling [79, 81, 83] can serve as an inspiration to pursue this goal. However, classic scheduling techniques prevent any concurrency between potentially conflicting transactions, whereas, in this case, it is only necessary to prevent concurrency between read-only transactions and the commit phase of update transactions.

Another limiting feature of Intel's HTM implementation, which prevents the use of

DMP-TM on these processors, is the lack of information about the address that triggered an access violation from within a HTM transaction. To overcome this obstacle, we foresee two possible directions: (i) the first consists in obtaining the address of the instruction that caused the exception via the Last Branch Records (LBRs), which in recent Intel processors store the last branches executed by the CPU and can be used to pinpoint the address of the instruction that caused an access violation [20] and (ii) a second approach is to investigate the use of Processor Tracing (PT), namely a recent ISA extension that supports inbuilt tracing mechanism for Intel TSX, and provides extensive control not only on the control flow within transactions, but also precise timing analysis on asynchronous events (like interrupts and signals).

The use of these techniques implies tackling non-trivial problems. The first approach rises the challenge of determining which memory address was targeted by the offending instruction (and not only the address of the offending instruction), based on the program control flow information stored in the LBRs. While for the second approach, although it appears that the information available using PT could be used to accurately estimate the memory addresses that triggered an access violation by a HTM transaction, the overheads incurred by tracing and analyzing this information in run-time are still unclear.

Bibliography

[1] R. R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, June 1997.

[2] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, Sept 2006.

[3] Andy Oram and Greg Wilson. *Beautiful Code*. O'Reilly, first edition, 2007.

[4] Ali-Reza Adl-Tabatabai ; Tatiana Shpeisman and Justin Gottschlic. *Draft Specification of Transactional Language Constructs for C++*.

[5] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[6] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC '06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[7] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[8] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

[9] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[10] Nuno Diegues and Paolo Romano. Time-Warp: Efficient abort reduction in transactional memory. *ACM Transactions on Parallel Computing*, 2(2):12:1–12:44, June 2015.

[11] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40:46–40:58, September 2008.

[12] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 265–274, New York, NY, USA, 2008. ACM.

[13] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54(4):70–77, April 2011.

[14] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the Power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 225–236, New York, NY, USA, 2013. ACM.

[15] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[16] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for IBM System Z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 25–36, Washington, DC, USA, 2012. IEEE Computer Society.

[17] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 29(2):6–16, March 2009.

[18] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 27–40, New York, NY, USA, 2010. ACM.

[19] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance

computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.

[20] Intel. Intel 64 and IA-32 architectures optimization reference manual. Technical report, Intel, 2016.

[21] IBM. Power ISA™ version 2.07b. Technical report, IBM, 2015.

[22] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM.

[23] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[24] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 39–52, New York, NY, USA, 2011. ACM.

[25] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 53–64, New York, NY, USA, 2011. ACM.

[26] Alexander Matveev and Nir Shavit. Reduced hardware transactions: A new approach to hybrid transactional memory. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 11–22, New York, NY, USA, 2013. ACM.

[27] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: A hybrid transactional memory for Haswell's Restricted Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 187–200, New York, NY, USA, 2014. ACM.

[28] Alexander Matveev and Nir Shavit. Reduced Hardware NOrec: A safe and scalable hybrid transactional memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 59–71, New York, NY, USA, 2015. ACM.

[29] Trevor Brown and Srivatsan Ravi. Cost of Concurrency in Hybrid Transactional Memory. In *LIPIcs 31st International Symposium on Distributed Computing*, volume 91 of *DISC '17*, pages 9:1–9:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[30] Luiz Andre Barroso and Urs Hoelzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[31] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, New York, NY, USA, 2010. ACM.

[32] Sanjay Ghemawat and Paul Menage. TCMalloc: Thread-Caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, 2005.

[33] `https://github.com/shadyalaa/POWER8TM`, 2017.

[34] `https://github.com/shadyalaa/GreenCM`, 2015.

[35] `https://github.com/pedroraminhas/DMP-TM`, 2018.

[36] Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[37] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[38] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, December 2007.

[39] T. D. Burd ; T. A. Pering ; A. J. Stratakos ; R. W. Brodersen. A dynamic voltage scaled microprocessor system. *IEEE Journal of Solid-State Circuits*, 35:1571 – 1580, Nov. 2000.

[40] **Shady Issa**, Pascal Felber, Alexander Matveev, and Paolo Romano. Extending Hardware Transactional Memory Capacity via Rollback-Only Transactions and Suspend/Resume. In *LIPIcs 31st International Symposium on Distributed Computing*, volume 91 of *DISC '17*, pages 28:1–28:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[41] Pascal Felber, **Shady Issa**, Alexander Matveev, and Paolo Romano. Hardware read-write lock elision. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys '16, pages 34:1–34:15, New York, NY, USA, 2016. ACM.

[42] **Shady Issa**, Pascal Felber, Alexander Matveev, and Paolo Romano. Extending Hardware Transactional Memory Capacity via Rollback-Only Transactions and Suspend/Resume. *Distributed Computing*, Submitted on January 15, 2018, currently under review.

[43] Pedro Raminhas, **Shady Issa**, and Paolo Romano. Enhancing efficiency of hybrid transactional memory via dynamic data partitioning schemes. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '18, pages 1–11, May 2018.

[44] **Shady Issa**, Paolo Romano, and Mats Brorsson. Green-CM: Energy efficient contention management for transactional memory. In *Proceedings of the 44th International Conference on Parallel Processing (ICPP)*, pages 550–559, Sept 2015.

[45] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[46] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Communications of ACM*, 14(10):667–668, October 1971.

[47] Peter A. Buhr, David Dice, and Wim H. Hesselink. High-performance n-thread software solutions for mutual exclusion. *Concurrency and Computation: Practice and Experience*, 27(3):651–701, 2014.

[48] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[49] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 340–347, New York, NY, USA, 1984. ACM.

[50] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing NUMA locks. *ACM Transactions on Parallel Computing*, 1(2):13:1–13:42, February 2015.

[51] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[52] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[53] Edward G. Coffman, Jr. and Peter J. Denning. *Operating Systems Theory.* Prentice Hall Professional Technical Reference, 1973.

[54] Posix.1-2008. http://pubs.opengroup.org/onlinepubs/9699919799/, 2013.

[55] Jonathan Corbet. Big reader locks. https://lwn.net/Articles/378911/, 2016.

[56] Stephen Hemminger. Kill big reader locks. http://git.kernel.org/cgit/linux/kernel/git/tglx/history.git/commit/?id=7f6634804cd6dd71dd9ea33aa9019c3870a1d423, 2003.

[57] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 219–230, Philadelphia, PA, June 2014. USENIX Association.

[58] Maya Arbel and Hagit Attiya. Concurrent updates with RCU: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014. ACM.

[59] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM.

[60] Paul E. McKenney and Jonathan Walpole. What is RCU, fundamentally? https://lwn.net/Articles/262464/, 2007.

[61] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[62] Ravi Sethi. Useless actions make a difference: Strict serializability of database updates. *J. ACM*, 29(2):394–403, April 1982.

[63] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[64] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC '06, pages 284–298, Berlin, Heidelberg, 2006. Springer-Verlag.

[65] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 226–236, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[66] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[67] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[68] Kevin E. Moore. *Log-based Transactional Memory.* PhD thesis, University of Wisconsin at Madison, Madison, WI, USA, 2007. AAI3278772.

[69] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009. ACM.

[70] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 3–14, New York, NY, USA, 2014. ACM.

[71] D. Castro, P. Romano, D. Didona, and W. Zwaenepoel. An analytical model of hardware transactional memory. In *IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '17, pages 221–231, Sept 2017.

[72] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel core, and POWER8. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 144–157, New York, NY, USA, 2015. ACM.

[73] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. *9th ACM SIGPLAN Wkshp. on Transactional Computing*, 2014.

[74] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 101–110, New York, NY, USA, 2010. ACM.

[75] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 258–264, New York, NY, USA, 2005. ACM.

[76] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management. In *Proceedings of the 19th International Symposium on Distributed Computing*, volume 3724 of *DISC '05*, pages 303–323, 2005.

[77] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust Contention Management in Software Transactional Memory. In *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL'05)*, 2005.

[78] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[79] Nuno Diegues, Paolo Romano, and Stoyan Garbatov. Seer: Probabilistic scheduling for hardware transactional memory. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 224–233, New York, NY, USA, 2015. ACM.

[80] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-Abort: Improving transactional memory performance through dynamic transaction reordering. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 4–18, Berlin, Heidelberg, 2009. Springer-Verlag.

[81] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 125–134, New York, NY, USA, 2008. ACM.

[82] Hugo Rito and João Cachopo. Props: A progressively pessimistic scheduler for software transactional memory. In *In the Proceedings of the 20th International Conference Euro-Par on Parallel Processing*, pages 150–161, Cham, 2014. Springer International Publishing.

[83] Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 7–16, New York, NY, USA, 2009. ACM.

[84] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 79–90, New York, NY, USA, 2010. ACM.

[85] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. TxLinux: Using and managing hardware transactional memory in an operating system. In *Proceedings of Twenty-first ACM*

SIGOPS Symposium on Operating Systems Principles, SOSP '07, pages 87–102, New York, NY, USA, 2007. ACM.

[86] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 169–178, New York, NY, USA, 2008. ACM.

[87] D. Rughetti, P. D. Sanzo, and A. Pellegrini. Adaptive transactional memories: Performance and energy consumption tradeoffs. In *2014 IEEE 3rd Symposium on Network Cloud Computing and Applications (ncca 2014)*, pages 105–112, Feb 2014.

[88] B. Goel, R. Titos-Gil, A. Negi, S. A. McKee, and P. Stenstrom. Performance and energy analysis of the restricted transactional memory implementation on haswell. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 615–624, May 2014.

[89] Nuno Diegues and Paolo Romano. Self-tuning Intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 209–219, Philadelphia, PA, June 2014. USENIX Association.

[90] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *J. Mach. Learn. Res.*, 3:397–422, March 2003.

[91] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[92] Diego Didona, Pascal Felber, Derin Harmanci, Paolo Romano, and Jórg Schenker. Identifying the optimal level of parallelism in transactional memory applications. *Computing*, 97(9):939–959, September 2015.

[93] Diego Rughetti, Paolo Romano, Francesco Quaglia, and Bruno Ciciani. Automatic tuning of the parallelism degree in hardware transactional memory. In *Euro-Par 2014 Parallel Processing*, pages 475–486. Springer International Publishing, 2014.

[94] Pierangelo Di Sanzo, Bruno Ciciani, Roberto Palmieri, Francesco Quaglia, and Paolo Romano. On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking. *Perform. Eval.*, 69(5):187–205, May 2012.

[95] Amin Mohtasham, Ricardo Filipe, and Joao Barreto. Frame: Fair resource allocation in multi-process environments. In *Proceedings of the IEEE 21st International Conference on Parallel and Distributed Systems*, ICPADS '15, pages 601–608. IEEE, 2015.

[96] Zhengyu He and Bo Hong. Modeling the run-time behavior of transactional memory. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '10, pages 307–315, Washington, DC, USA, 2010. IEEE Computer Society.

[97] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, pages 719–728, Berlin, Heidelberg, 2008. Springer-Verlag.

[98] Amin Mohtasham and João Barreto. RUBIC: Online parallelism tuning for co-located transactional memory applications. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 99–108, New York, NY, USA, 2016. ACM.

[99] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. ProteusTM: Abstraction meets performance in transactional memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 757–771, New York, NY, USA, 2016. ACM.

[100] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[101] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008.

[102] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A benchmark for software transactional memory. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM.

[103] TPC Council. Transaction Processing Performance Council, TPC BENCHMARK™ C. Revision 5.11. February 2010.

[104] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[105] E. Jones. Stand-alone in-memory TPC-C implementation. `https://github.com/evanj/tpccbench`, 2017.

[106] Brad Fitzpatrick. Distributed caching with Memcached. *Linux J.*, 2004(124):5–, August 2004.

[107] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing legacy code: An experience report using GCC and Memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 399–412, New York, NY, USA, 2014. ACM.

[108] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8:1–8:14, Jan 2015.

[109] Dave Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 21–33, Washington, DC, USA, 2007. IEEE Computer Society.

[110] T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 1985.

[111] D. Berry and B. Fristedt. Bandit problems. *London: Chapman and Hall*, 1985.

[112] Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. A transactional memory with automatic performance tuning. *ACM Transactons on Architecture and Code Optimaztion*, 8(4):54:1–54:23, January 2012.

[113] Yehuda Afek, Amir Levy, and Adam Morrison. Programming with hardware lock elision. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 295–296, New York, NY, USA, 2013. ACM.

[114] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir. Hardware extensions to make lazy subscription safe. *CoRR*, abs/1407.6968, 2014.

[115] Dan Nussbaum, Yosef Lev, and Mark Moir. PhTM: Phased transactional memory. In *The Second ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT '07, New York, NY, USA, 2007. ACM.

[116] Wenjia Ruan and Michael Spear. Hybrid transactional memory revisited. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363*, DISC '15, pages 215–231, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[117] Torvald Riegel, Christof Fetzer, and Pascal Felber. Automatic data partitioning in software transactional memories. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 152–159, New York, NY, USA, 2008. ACM.

[118] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 185–196, New York, NY, USA, 2009. ACM.

[119] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, September 2010.

[120] João Paiva, Pedro Ruivo, Paolo Romano, and Luís Rodrigues. AUTOPLACER: Scalable self-tuning data placement in distributed key-value stores. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 119–131, San Jose, CA, 2013. USENIX.

[121] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero. Clock gate on abort: Towards energy-efficient hardware transactional memory. In *Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, May 2009.

[122] Alexandro Baldassin, Joao P. L. de Carvalho, Leonardo A. G. Garcia, and Rodolfo Azevedo. Energy-performance tradeoffs in software transactional memory. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '12, pages 147–154, Washington, DC, USA, 2012. IEEE Computer Society.

[123] A. Branover, D. Foley, and M. Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 32(2):28–37, March 2012.

[124] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-management architecture of the Intel® microarchitecture code-named Sandy Bridge. *IEEE Micro*, 32(2):20–27, March 2012.

[125] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. The TURBO diaries: Application-controlled frequency scaling explained. In *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC '14, pages 193–204, Philadelphia, PA, June 2014. USENIX Association.

[126] Cesare Ferri, Samantha Wood, Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Embedded-tm: Energy and complexity-effective hardware transactional memory for embedded multicore systems. *Journal of Parallel and Distributed Computing*, 70(10):1042 – 1052, 2010. Transactional Memory.

[127] Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Energy reduction in multiprocessor systems using transactional memory. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, pages 331–334, New York, NY, USA, 2005. ACM.

[128] Epifanio Gaona-Ramirez, Ruben Titos-Gil, Juan Fernandez, and Manuel E. Acacio. Characterizing energy consumption in hardware transactional memory systems. In *Proceedings of the 22nd International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '10, pages 9–16, Washington, DC, USA, 2010. IEEE Computer Society.

[129] Jons-Tobias Wamhoff, Christof Fetzer, Pascal Felber, Etienne Rivière, and Gilles Muller. Fastlane: Improving performance of software transactional memory for low thread counts. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13. ACM, 2013.

[130] Cesare Ferri, Ruth Iris Bahar, Mirko Loghi, and Massimo Poncino. Energy-optimal synchronization primitives for single-chip multi-processors. In *Proceedings of the 19th ACM Great Lakes Symposium on VLSI*, GLSVLSI '09, pages 141–144, New York, NY, USA, 2009. ACM.