Enhancing efficiency of Hybrid Transactional Memory via Dynamic Data Partitioning Schemes

Pedro Luís Galvão Raminhas

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Paolo Romano

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Paolo Romano
Member of the Committee: Prof. António Manuel Ferreira Rito da Silva

October 2016
Acknowledgements

I cannot start without thanking my advisor, Paolo Romano, for the support given throughout this thesis. For every meeting and hour after hour discussing possible ways to solve problems and for his patience and confidence whenever we had to face a problem. His dedication for research is so contagious that I think I will carry it with me for the rest of my life. Thanks for the opportunity to work with you.

Also, this thesis could not been done without Shady Alaa. For that, I say a big thank you, for all the hours passed on Skype debugging code or just trying to solve problems that inevitably appear. The majority of the knowledge that I carry on about Transactional Memory and Concurrent Programming was passed by him, with his patience he made the learning curve of Transactional Memory less steeper.

I would like to thank my family, specially my parents, for supporting me in all steps of my life and making sure that I could be whatever I dream of, this chapter of my life was one more that I could count on them.

Also, I want to thank all my friends for the support given throughout this journey. A big thank you to Leonardo Cavaco for always being on the loop about all what is happening on my thesis and to share with me long hours in the coffee place just discussing a lot of issues and trying to decompress to later be more focused on work.

Last, but not the least, I cannot finish without a special thank you to Ana Caldeira. With her strength and belief in my capabilities, she showed me that I can accomplish more than I can ever dream of. She taught me how to chase my dreams, whatever they are. She was my companion throughout this thesis, sharing all the victories and losses with me. I cannot but feel so blessed to have such an inspiring and yet so supportive person in my life. Thank you for being present in all times.

This work was partially supported by Portuguese funds through Fundacão para a Ciência e Tecnologia via project SATURN (PTDC/EEISCR/1743/2014).

Lisboa, October 2016
Pedro Luís Galvão Raminhas
To my parents, Fátima and Paulo.
A memória transacional é um paradigma poderoso que promete simplificar o desenvolvimento de programas paralelos, enquanto permite obter desempenho semelhante ao dos mecanismos de locking. Na memória transacional, os programadores apenas têm de especificar que partes do código de um programa têm de ser executadas de forma atômica, e não como esta deve ser atingida.

Este documento primeiro apresenta uma análise do estado da arte da memória transacional, analisando algumas das principais implementações da memória transacional baseada em software (STM), baseada em hardware (HTM), ou uma combinação dos dois (Hybrid TM).

Com base na análise crítica do estado da arte, este documento propõe um novo algoritmo para uma implementação da memória transacional híbrida, a qual chamamos DMP-TM, que visa minimizar as perdas de desempenho devido à sincronização entre HTM e STM, apoiando-se num novo mecanismo de particionamento dinâmico de memória. A ideia chave por detrás de DMP-TM é depender do mecanismo de proteção dos sistemas operativos para determinar partição de memória, nas quais HTM e STM podem operar, visto não existir interferência mutua. Este desenho do sistema contrasta com o desenho das implementações da Hybrid TM correntes, que possuem perdas de desempenho devido à sincronização dos metadados.

Avaliamos DMP-TM através de benchmarks sintéticas, bem como benchmarks base na avaliação para a memória transacional. O nosso estudo experimental mostra que DMP-TM pode aumentar até 4x o desempenho comparado com outras implementações HTM e STM e até 10x o desempenho de outras implementações da Hybrid TM.

**Palavras Chave:** Programação Paralela, Memória Transacional, Partição de memória, Eficiência, Memória Transacional Híbrida.
Abstract

Transactional Memory (TM) is a powerful paradigm that promises to simplify the development of concurrent programs, while still achieving performance similar to locking. With TM, programmers only need to specify which parts of the code have to appear as executed atomically, and not how atomicity should be achieved.

This document first overviews the state of the art of TM, analyzing some of the main TM implementations based on software (STM), hardware (HTM), or on combination thereof (Hybrid TM).

In the light of the critical analysis of the state of art, this document proposes a novel Hybrid TM algorithm, which we call DMP-TM, that aims to minimize the synchronization overheads between HTM and STM by relying on a novel dynamic memory partitioning scheme. The key novel idea underlying DMP-TM is to leverage on operating system level memory protection mechanism to enforce dynamic memory partitions, in which HTM and STM can operate assuming no mutual interference. This design contrasts with current state of the Hybrid TM designs, which incur extra-overheads by checking the metadata.

We evaluated DMP-TM using both synthetic and standard benchmarks for Transactional Memory. Our experimental study highlights that DMP-TM can achieve up to 4x time speedups when compared to state of the art HTM and STM and up to 10x speedups compared with state of the art Hybrid TM.

Keywords: Parallel Programming, Transactional Memory, Memory Partitioning, Efficiency, Hybrid Transactional Memory
# Contents

1 Introduction ................................................................. 2
   1.1 Contributions ......................................................... 3
   1.2 Structure of the Document ........................................... 4

2 Related Work .................................................................. 5
   2.1 Software Transactional Memory ........................................ 5
      2.1.1 TinySTM .......................................................... 6
      2.1.2 SwissTM .......................................................... 7
      2.1.3 NOrec .............................................................. 8
   2.2 Enhancing efficiency of TM systems ................................. 9
      2.2.1 STM with Data Partitioning Scheme ........................... 9
      2.2.2 Providing strong atomicity using operating system’s page protection mechanism ........................................... 9
   2.3 Hardware Transactional Memory ...................................... 10
      2.3.1 Lazy Subscription ............................................... 12
   2.4 Hybrid Transactional Memory ......................................... 13
      2.4.1 Hybrid NOrec .................................................... 13
      2.4.2 Hybrid-LSA ....................................................... 15
      2.4.3 Hybrid NOrec-2 .................................................. 16
      2.4.4 Invyswell .......................................................... 17
      2.4.5 Reduced Hardware NOrec ....................................... 18
   2.5 Benchmarks for TM ..................................................... 18
      2.5.1 STMBench7 ......................................................... 19
      2.5.2 Lee-TM ............................................................ 19
      2.5.3 STAMP ............................................................. 19
      2.5.4 Memcached ......................................................... 20
      2.5.5 Kyoto Cabinet ..................................................... 20
# Dynamic Partitioning Hybrid TM

3.1 High Level Overview ............................... 22

3.1.1 Memory Protection .......................... 22

3.1.2 Mapping of Memory ........................... 23

3.1.3 HTM Component ............................... 25

3.1.4 STM Component ............................... 25

3.1.5 Signal Handler and Contention Manager .......... 25

3.2 Detailed description ............................. 27

3.2.1 API ........................................... 27

3.2.2 Operating system mechanisms used for memory mapping and protection .... 27

3.2.3 Synchronization of HTM and STM component ........ 28

3.2.3.1 Page Metadata ............................ 28

3.2.4 Signal Handler ................................. 29

3.2.5 Contention Manager ........................... 30

3.2.6 STM Algorithm ................................ 31

3.2.6.1 Reads performed by STM transactions ......... 31

3.2.6.2 Writes performed by STM transactions ........ 31

3.2.6.3 Restart of an STM transaction ............... 32

3.2.6.4 Commits performed by transactions running STM ........ 33

3.2.7 Optimizations ................................ 33

3.2.7.1 Bitmap representing pages read and written .... 33

3.2.7.2 Usage of a Global transition count ............ 34

3.2.8 Variants of the algorithm ...................... 34

3.2.8.1 Fallback to Single Global Lock ............... 34

3.2.8.2 Fallback to TinySTM ....................... 35

3.2.8.3 Falling back to single global lock acquired by hardware transactions .... 35

3.2.8.4 Falling back to Global lock using memory barriers ........ 35

3.2.9 Memory Allocation ............................ 36

3.3 Pseudocode of the solution ...................... 37

3.3.1 Signal Handler Pseudocode ................... 37
# List of Figures

3.1 Map of the address space in the HTM Heap and in the STM Heap ................. 24
3.2 Transaction running STM performs a read operation .......................... 24
3.3 Scenario where STM and HTM transactions abort each other ............... 30
3.4 Transition count changes and aborts STM transaction ....................... 32
3.5 Transaction running STM issues a write operation .......................... 33
4.1 Comparison of different variants of the algorithm using the synthetic benchmark with time to change partitions=\infty ................................. 49
4.2 Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using probability of accessing the STM pool = 0% .................. 51
4.3 Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using probability of accessing the STM pool = 2% .................. 52
4.4 Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using probability of accessing the STM pool = 5% .................. 54
4.5 Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using probability of accessing the STM pool = 50% ............ 55
4.6 Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using probability of accessing the STM pool = 100% .......... 56
4.7 Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using hashmap with partition change ....................... 57
4.8 Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using hashmap with partition change of 1 and 0.5 seconds .......... 61
4.9 Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using vacation benchmark .......................... 62
4.10 Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using Genome benchmark .......................... 63
4.11 Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using intruder benchmark .......................... 64
List of Tables

2.1 Transactional store and load of the state of the art processors that successful implement HTM .................................................. 12

2.2 Comparison of various TM state of the art implementations: TinySTM, SwissTM, NOrec, TSX HLE, TSX RTM with global lock, IBM Power8, Hybrid NOrec, Hybrid-LSA, Hybrid NOrec-2, Invyswell, Reduced Hardware NOrec and the two proposed solutions ........................................... 14

4.1 Breakdown of commit types of state of art TM implementations with the probability of accessing a page in the STM pool=2% with $p_{op} = 10\%$ .................................................. 53

4.2 Breakdown of commit types of state of art TM implementations with the probability of accessing a page in the STM pool=2% with $p_{op} = 10\%$ with pages changing pool every 2 seconds ............................................... 55

4.3 Breakdown of commit types of state of art TM implementations with the probability of accessing a page in the STM pool=2% with $p_{op} = 50\%$ with pages changing pool every 2 seconds ............................................... 58

4.4 Breakdown of commit types of state of art TM implementations with the probability of accessing a page in the STM pool=2% with $p_{op} = 90\%$ with pages changing pool every 2 seconds ............................................... 58

4.5 Breakdown of commit types of state of art TM implementations running vacation low .................................................. 59

4.6 Breakdown of commit types of state of art TM implementations running vacation High .................................................. 59

4.7 Breakdown of commit types of state of art TM implementations running vacation with different number of queries per client .................................................. 60

4.8 Breakdown of commit types of state of art TM implementations running Genome .................................................. 61

4.9 Breakdown of commit types of state of art TM implementations running Intruder .................................................. 62
Acronyms

**AV**  Access Violation

**CPU**  Central Processing Unit

**DMP**  Dynamic Memory Partitioning

**HTM**  Hardware Transactional Memory

**HyTM**  Hybrid Transactional Memory

**OS**  Operative System

**SGL**  Single Global Lock

**STM**  Software Transactional Memory

**TM**  Transactional Memory
Introduction

Multicore processors have become the standard architecture in today's computing systems, ranging from high-end servers, to personal computers, as well as smartphones and embedded systems, all have the multicore technology and the trend is to increase the number of cores towards manycore architectures. Unfortunately, the development of parallel applications that can fully exploit the advantages of the multicore technology is far from being a trivial task.

Indeed, when using traditional lock-based synchronisation mechanisms, developers are faced with a major dilemma. They can either opt for using coarse-grained locks, which simplifies significantly reasoning on the correctness of concurrent applications, but can severely hinder parallelism and hamper performance. Or they can opt for a second alternative which is to use fine-grained locking. This approach allows for extracting the most parallelism possible, avoiding the inefficiency of coarse-grained locking, but it is harder to devise and more prone to deadlocks and livelocks, which are notoriously hard to debug [34]. In fact, fine grained locking is considered difficult to use for the average programmer [34], who may not be necessarily well trained for the development of concurrent applications.

Transactional Memory (TM) [26] precisely answers this urge by specifying a new programming paradigm that alleviates the complexity of classical locking mechanisms while ensuring performance similar or even better than complex fine-grained locking schemes. TM borrows the concept of transactions from databases and applies it to parallel programming: with TM, programmers devise code into blocks that will be ensured by the TM system to run atomically. The underlying system is responsible for either committing the transactions, and making its modifications globally visible, or to aborting the transactions and ensuring that no modifications are observed out of the context of the transaction. This approach leads to drastically simplifying the development of parallel applications and abate their time to market and costs.

Transactional Memory was initially proposed twenty years ago as an extension to multi-processors’ cache coherence protocols [26]. Due to the difficulty of rapid prototyping in hardware environments, researchers resorted to Software Transactional Memory to advance the state of the art [13; 20; 21; 23]. Simultaneously, hardware-based implementations have also been proposed, whose designs were validated using simulators.

Hardware Transactional Memory (HTM) is currently supported by the mainstream Intel Haswell and the IBM Power8 processors. These implementations are best-effort, which means that they always ensure correctness, but do not provide any progress guarantee. This means that transactions executing in hardware may never commit (even in absence of concurrency) due to inherent limitations of hardware, such as transaction exceeding capacity of hardware or running prohibited instructions. This raises the need for a software fallback path to ensure forward progress [19].
This fallback path can be either as simple as a global lock or as complex as a software transactional memory, which is designated by Hybrid Transactional Memory (Hybrid TM) [9; 12; 14; 30; 37]. The goal of Hybrid TM is to fully exploit best-effort HTM as much as possible, and in the case of abort, fall back to the more costly Software Transactional Memory (STM), which provides progress guarantee.

Despite the number of papers published in this area in recent years, Hybrid TM still suffer from large overheads [19]. These overheads are particularly exacerbated on Hybrid TMs which require HTM to manipulate the per-location metadata (often referred to as Ownership Records, ORecs) used by STM, which extends significantly the working set of HTM transactions making them prone to capacity exceptions but also in Hybrid TMs that use a software fallback without per location metadata, as these Hybrid TMs suffer from a scalability bottleneck as every transaction must read a sequence lock. Further, they can induce spurious aborts of HTM transactions if any concurrent, non-conflicting HTM commits [19].

Benchmarks’ tests indicate that the performance of state of the art Hybrid TM can not surpass HTM performance for workloads characterised by short transactions with small read and write-set, and that it can not surpass the scalability of STM for workloads characterised by long transactions with large read and write-set. Results also show that Hybrid TM has costs incurred by synchronising both the execution of HTM and STM, as the memory locations accessed by one TM has to be validated with the read and write-set of both TMs. This led us to conclude that the performance of Hybrid TM is highly dependent on the workload.

The goal of this work is to provide a new class of Hybrid TM without the main drawback of state of the art Hybrid TM: the overheads incurred by the synchronization of transactions running HTM and transactions running STM. To accomplish that, we rely on operating system’s memory protection mechanism to dynamically establish a memory partitioning scheme that ensures that HTM and STM execute, at any time, on disjoint memory regions. By delegating to the Operative System’s (OS) memory protection mechanism the task of detecting violations of the currently established memory partitioning scheme, DMP-TM investigates an interesting trade-off, which has not currently explored in the literature: reducing the runtime overheads for detecting conflicts among STM and HTM transactions, at the cost of a large performance penalty in case conflicts between STM and HTM transactions do materialize.

As it will be shown via an extensive experimental study, this design choice allows DMP-TM to achieve up to 4x speedups compared to HTM and STM, and 10x speedups compared to Hybrid-NOrec [12], in workloads where the memory regions accessed by HTM and STM transactions are unlikely to overlap.

1.1 Contributions

This dissertation provides the following contributions:

- Performs a critical analysis of the state of art in the area of Transactional Memory, which serves as a motivation for the Hybrid TM solution herein proposed.
• Introduces DMP-TM, a novel Hybrid TM algorithm that departs from conventional Hybrid TM algorithms by relying on operative system based memory protection mechanisms to ensure safe coexistence between transactions concurrently executing in hardware and software.

• Reports the results of an extensive experimental evaluation, based both on synthetic benchmarks aimed to investigate the performance tradeoffs underlying the proposed design, as well as standard benchmarks in the TM literature.

1.2 Structure of the Document

The rest of this document is organized as follows.

Chapter 2 provides an analytical review of the state of art of Transactional Memory, namely Software Transactional Memory, Hardware Transactional and Hybrid Transactional Memory, presenting and comparing its advantages and disadvantages. Also, it presents an overview of the state of art of TM benchmarks used to evaluate the TM implementations.

Chapter 3 represents the core of this thesis, it presents a novel Hybrid TM implementation, named Dynamic Memory Partitioning (DMP-TM). Starts by a high level overview of the implementation, presenting how we synchronize both HTM and STM. Then, it presents a detailed description of the solution accompanied by the pseudocode of the solution.

Chapter 4 presents an extensive experimental evaluation, based on both synthetic benchmarks and several standard benchmarks of the TM literature.

Finally, Chapter 5 concludes this dissertation, by resuming the key results achieved and discussing possible directions for future work.
This section overviews the state of the art of Transactional Memory, it first begins with STM and presents three implementations that address different design approaches. Then, it presents a set of papers intended to enhance the efficiency of STM, either by allowing the execution of non-transactional code or by using memory partitions to allow different design optimisations for each partition and achieve the better throughput than the base STM. Then, it presents the state of the art of HTM and shows the virtues and limitations of each HTM available. Next, it presents the state of the art of Hybrid TM and it presents six different approaches to synchronise both HTM and STM as well as a table that compares the characteristics of the state of the art Hybrid TM with the proposed solution. Finally, is presented an overview of state of the art benchmarks used to evaluate TM.

2.1 Software Transactional Memory

Software Transactional Memory (STM) is a category of systems that implement the Transactional Memory abstraction through a purely software-based runtime library.

During the last decade, STM was object of a thorough research mainly because it is a solution that is not bound to a specific architecture, nor it is restricted by the underlying hardware, making STM a widely portable solution. In STM, transactional reads and writes are tracked by a software runtime, which has the responsibility of maintaining the read and write set of transactions. Besides maintaining the read and write sets, most STMs keep additional metadata about transactions, such as locking, ownership records and versioning.

Existing literature in the area of STM can be coarsely classified according to the following design choices: word-based versus object-based, optimistic vs pessimistic conflict detection and write-through versus write-back.

- Word-based versus Object-based - word-based STMs access the memory directly at the granularity of machine words or larger chunks of memory, yet object-based STMs access the memory at object granularity and it requires the TM to be aware of the object associated in every access.

- Pessimistic versus Optimistic conflict detection - STMs that use pessimistic conflict detection rely upon a set of locks to control the concurrency between multiple transactions. Whenever a conflict is detected, the transactions that caused the conflict are immediately aborted, which minimizes further conflicts with on-going transactions. On other side, STMs that use optimistic conflict detection only detect conflicts at commit-time since read-write conflicts, that are common between long running transactions and short ones should not incur the abort and further waste of work of the
longer transactions, rather delay the conflict until commit-time and check which transaction should abort.

- Write-through vs Write-back - STMs can be write-through, writing their updates directly to memory and storing the previous ones on a undo log, or write-back which means that changes are only written to memory at commit-time.

The best design can vary according to a number of factors like the underlying architecture, the number of cores per CPU and the size of caches. More important to note is that the efficiency that will result from these choices is highly dependent from the type of workload generated by the application.

In comparison with HTM, STM systems incur heavy instrumentation on the code, issuing modification on the atomic code blocks in order to account for transactional operations and other operations performed by the underlying system as log keeping and conflict detection. Despite of the drawbacks, state of the art STM normally provides stronger progress guarantees than HTM: being fully implemented in software, STM does not suffer from the restrictions that affect HTM and can support the execution of arbitrarily long transactions, i.e., transaction in which the read and write-set does not fit in the cache (see table 2.1)

### 2.1.1 TinySTM

TinySTM is a STM implementation presented by Felber et al. in 2008 [21].

Authors state that the characteristics of the workload of an STM implementation plays a major role in selecting the right choice and configuration parameters. Some of the characteristics are the ratios to update read-only transactions, the size of read and write-sets and contention on shared memory.

TinySTM is a word-based STM implementation, meaning that it allows for directly mapping of transactional accesses to the underlying memory subsystem.

This STM uses an encounter-time locking mechanism, as authors state that it increases the transactions throughput because transactions do not perform useless work. This mechanism also allows the efficient handling of reads-after-write conflicts without requiring expensive or complex mechanisms, a valuable feature especially when write-sets have non-negligible size.

Along with encounter time locking, two other strategies can be used in TinySTM to access memory: write-through and write-back access. In write-through access, transactions immediately write to memory and undo updates if aborted. In write-back access, transactions do not update until the commit time.

As most word-based STM designs, TinySTM relies upon a shared array of locks to manage concurrent accesses to memory. Each lock covers a portion of the address space. Addresses are per-stripe mapped to the lock based on a hash function. Each lock represents a size of address in memory, as the least significant bit shows whether lock is owned or not. If the lock is owned, the remaining bits of address store the owner transaction, if not, a version number based on the timestamp of the last owner transaction is stored in the remaining bits.

The locks are used to protect memory, when a transaction issues a read it first checks if the lock protecting the read item is currently being held by other transaction. Then, it reads the value of the item,
and finally the lock again. If the lock is not held by other transaction or the value of the lock did not change between both reads, then the value read is consistent.

When a transaction writes to a memory location, it reads the lock entry from selected memory addresses. If it finds that the lock bit is set then it verifies either the current transaction is the owner or not. If current transaction is the owner, then it simply writes the new value to memory location. If it is not owner of the current transaction then the current transaction can wait for some time to get resources free or aborts immediately.

TinySTM guarantees that there is a consistent read-set upon each read for read-only transactions, therefore read-only transactions do not need to validate their read-sets upon commit. Update transaction have to validate their read-set before updating any memory location. The downside of this approach is that the validation of large read-set may be costly, to avoid this overhead the number of locks can be reduced. On the other hand reducing the number of locks can increase abort rate as it increases false sharing.

In order to solve this problem, authors proposed hierarchical locking. In addition to the shared array of locks, it is maintained a smaller array of counters which goal is to hold the number of commits done to locations in that region. An hash function is used to map memory addresses to counters, this function is consistent with the one used to map addresses to lock arrays, i.e., memory locations that are mapped to the same lock are also mapped to the same counter, which implies that a counter covers multiple locks and the associated memory addresses. This scheme allows transactions to determine whether locks have been acquired or not, however it has an overhead associated with it but authors state that it is amortised when transactions have large read sets or when there are few writes from competing transactions.

TinySTM periodically adapts the tuning parameters at runtime and measures the throughput over a period of approximately one second. The system registers the most recent throughput for each tuning configuration. Each tuning configuration is a triple consisting of the number of locks, the number of shifts, and the size of the hierarchical array. The tuning strategy is a hill climbing algorithm with a memory and forbidden areas in order to achieve the best configuration to the given workload.

### 2.1.2 SwissSTM

SwissSTM is a STM implementation presented by Dragojevic et al. in 2009 [20].

Authors state that state of the art STMs only have good performance for workloads with small scale transactions, but in practice do not work well for large scale applications like business software and video games.

This STM uses four-word lock granularity to protect shared memory, as authors state that this approach outperforms both word-level and cache line-level locking for all benchmarks considered.

It uses a mixed invalidation conflict detection scheme which eagerly detects write/write conflicts. Thus preventing long transactions, that are doomed to abort, to continue their execution and possibly cause more conflicts with other running transactions. On the other hand, read/write conflicts, commonly caused by short transaction with longer ones, are lazily detected. By using invisible reads and
allowing transactions to read objects acquired for writing, SwissTM detects read/write conflicts late, thus increasing inter-transaction parallelism. A time-based scheme is used to reduce the cost of transaction validation with invisible reads.

For conflict detection it uses a two-phase contention manager scheme that incurs no overhead on read-only and short read-write transactions. Upon a write, every transaction increments a local counter. If the value of this counter is below 10, then it is considered a short transaction and in case of conflict it will use a timid contention management scheme, aborting on the first encountered conflict. Transactions that are more complex increment the global counter above 10 are then switched dynamically to the Greedy mechanism that involves more overhead but favours these transactions, thus preventing starvation. This means that more complex transactions have higher priority than simpler transactions. Additionally, transactions that abort due to write/write conflicts back-off for a period proportional to the number of their successive aborts, hence reducing the probability of aborting repeatedly because of the same conflict.

Authors state that SwissTM outperforms all state of the art STM in mixed workloads characterised by non-uniform, dynamic data structures and various transaction sizes while also delivering good performance in smaller-scale scenarios.

2.1.3 NOrec

NOrec is a STM library presented in 2010, by Luke Dalessandro, Michael Spear and Michael Scott [13].

Authors advocate a minimal approach on STM handling of metadata. NOrec does not need fine grained, shared metadata, only requires a single global sequence lock (seqlock) for concurrency control. Transactions buffer writes and log read address/value pairs in thread local structures.

Upon start, transactions poll back seqlock and check if there is any transactions in write-back phase; if there is, transactions waits until there is no transaction in the write-back phase. Committing writers increments the seqlock in order to signal that they have written new values to memory.

When a read is issued, active transactions poll the global seqlock and compare it with the one held locally by the transaction. A new value is evidence of possible inconsistency and triggers validation, which is carried out in a value-based style by comparing the pair address/value in log to the actual values in main memory.

In this protocol, only a single writer can commit and perform writeback at a time. This sequential bottleneck is minimised by validation prior to acquiring the seqlock for commit.

By successfully incrementing the seqlock, a writer transitions to a committed state in which it immediately performs its writeback and then releases the lock. Read-only transactions can automatically commit because their reads are proven to be consistent because of the validation mechanism.

NOrec scales well when its single-writer commit serialisation does not represent the overall application bottleneck, i.e., writeback does not dominate the runtime, and has been shown to have low latency, although its read set entries are twice as large as the corresponding orec-based implementations, as NOrec stores both the address of the read location and the value that was seen.
NOrec guarantees consistency among transactional and non-transactional accesses to shared data, a characteristic named privatization [31; 40], which is required for compatibility with the C++ draft TM standard [5]. Further, it provides support for closed nesting, where inner transactions can abort and restart separately from their parents, which is expected to improve performance in certain applications.

Authors conclude that NOrec have low overhead and high scalability, which is ideal for use in legacy system as well as a fallback mechanism in Hybrid TM due to its robust performance.

2.2 Enhancing efficiency of TM systems

Accompanying the development of STM implementations, there have been a significant effort by the researchers to improve the efficiency of TM systems. In this section, we present two papers that fall out of the categories defined in this dissertation as both try to improve the efficiency of existing STM implementations.

2.2.1 STM with Data Partitioning Scheme

In order to mitigate the concern of different parts of the data having different access patterns, thus requiring specific design optimisations, Riegel et. al [36] proposed the usage of a data partitioning scheme integrated with TinySTM [21]. This approach enables STM to divide data structures in partitions and to compose different specialised optimisations for each partition accordingly to the workload or even to use a different STM backend in that partition, hence improving the transactional throughput of each partition.

To identify the partitions of an application, it is constructed a Data Structure (DS) graph at compile-time for every function encountered in the program. The DS graph is determined by analysing to which node (partition) does the pointers in a program are permitted to point to, and whenever two pointers stored in the same field point to disjoint nodes, those nodes are unified. Each partition has a type that indicate the concurrency control. The partitions types vary from read-only partitions to partitions with multiple locks, each partition type is determined at runtime accordingly to the number of aborts.

2.2.2 Providing strong atomicity using operating system’s page protection mechanism

In order to provide strong atomicity in an implementation of transactional memory, i.e., safe access between transactional and non-transactional code to same locations, Abadi et. al [4] propose the usage of operating system’s page protection mechanism to regulate the shared access of transactional and non-transactional code.

In order to detect conflicts, the process's virtual space is organized in a way that its heap is mapped twice, one heap is used by the non-transactional code and the other heap is used by code executed inside of a transaction. By using this organization of the heap in addition to the usage of the operation
system's page protection is possible to prevent the access of non-transactional code by changing the access right of the page in the virtual address space of its heap.

Since changing the access rights of a certain page implies a non-negligible cost, Abadi et al also proposed a set of dynamic code updates intended to reduce the high cost of AVs in off-the-shelf hardware.

2.3 Hardware Transactional Memory

Although TM was originally proposed as an micro-architectural feature [26], the absence of hardware support switched a large body of TM research into STM [13; 20; 21; 23]. Unlike Hardware solutions, STM require instrumentation of reads and writes in order to track the read and write-set of active transactions and detect conflicts between them. This instrumentation can introduce, in certain scenarios, large overheads that are non-negligible and can hinder performance with respect to conventional fine-grained locking.

HTM is thus desirable, as it relies on modified cache coherence protocol in order to achieve atomicity and isolation. The main advantage of HTM is that it requires no instrumentation of reads and writes, which degrades the performance of STM solutions, as the read and write-set are stored in L1 cache and conflicts are detected by the cache coherence protocol. A major drawback is that current HTM does not guarantee that a transaction will succeed, even without concurrency, due to its limited nature, which is limited by the fact that read and write-set fits in L1 cache.

The maturing of TM led to a change in the industry of processor manufacturing, as manufacturers revealed the first processors that successfully implemented HTM: Azul [11] and Rock [18] processors. Unfortunately, these system were not usable, as Azul programming interface was not disclosed, thus making HTM hidden for the programmers and Rock processor was cancelled before reaching the market. IBM Blue Gene/Q was the first processor on the market that provide an HTM implementation [41], followed by eEnterprise EC12 [2], IBM Power8 [3] and Intel Haswell [42]. This represented a significant milestone for TM, as with Intel Haswell processor, TM became available on commodity hardware from high-end servers to mainstream laptops.

Haswell uses the L1 cache for conflict detection and maintaining versions and transactional metadata. However, the details about the conflict detection and transaction capacities have not been disclosed by Intel. The Intel Transactional Synchronization Extensions (TSX), i.e., the code-name for the HTM API in Haswell CPUs, comprises two possible interfaces, Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). The former allows to elide locks [7] and execute code speculatively in a backwards-compatible manner. RTM leverages on the same hardware as HLE, but exposes a transaction interface in which the programmers only need to specify which blocks are atomic, leaving to the underlying system the responsibility to ensure the correct execution of transactions.

A very important detail is that the begin instruction requires to specify a software handler to deal with transaction aborts, and thus provide progress guarantee. The software fallback must co-operate with HTM in order to ensure correctness and the mechanism used has a great impact on the performance of TSX. The simplest approach and the one suggested by Intel's optimisation manual is the usage of a
single global lock as a fallback mechanism. When a hardware transaction aborts, it has the alternative to check and acquire the global lock if it is free. To ensure correctness, every hardware transaction must subscribe to the global lock and ensure it is free, as the underlying transactional semantics will guarantee that a transaction only commits when there is no ongoing fallback execution.

IBM Blue Gene/Q uses the L2 cache for conflict detection and updates buffering. It assigns a unique speculation ID to each transaction and uses the L2 cache to store the speculation ID upon a transactional access. IBM Blue Gene/Q has two transactional execution modes: a short-running mode and a long-running mode. In the short-running mode, it uses only the L2 cache to buffer transactional data, and in the long-running mode, it uses the L1 cache to buffer some of the transactional data though it invalidates all of the L1 cache lines at the start of each transaction.

IBM zEC12 uses the L1 cache for both conflict detection and loads buffering, along with a special LRU-extension, which function is to record the evicted cache lines. The transactional stores are buffered in an 8-KB gathering store cache, which is private for each processor and is located between the L1 cache and the L2/L3 caches. IBM zEC12 also provides constrained transactions, which are transactions that are guaranteed to commit.

IBM Power8 uses content-addressable memory (CAM) linked with the L2 cache for conflict detection [33]. The L2 TMCAM records the cache-line addresses that are accessed in the transactions with bits to represent read and write. Although the transactional stored data is buffered in the L2 cache, the transaction capacity is bounded by the size of the L2 TMCAM. Power8 supports rollback-only transactions, which are transactions that support normal transactional semantics, i.e., store buffering, but not conflict detection. Also, Power8 supports the usage of suspend/resume operations, which allows to escape from a transactional context and access a variable without risking to incur data conflicts.

Nataike et. al in [33] compared all state of the art processors that provide HTM, by using a modified version of the STAMP’s benchmark suite [32] better suited to accommodate the limited capacity of HTM. The experimental results showed that there is no single HTM system that is more scalable than the others in all of the benchmarks. Each HTM system has its own implementations issues that limit the scalability in certain workloads.

Results shows that Intel Core has load and store capacity of 4MB and 22KB respectively. This processor has extra transaction aborts due to hardware prefetching, as it could raise a conflict that would not exist if the hardware did not prefetch other cache lines in advance. L1 and L2 size are respectively 32KB and 256 KB.

IBM Power8 has more capacity-overflow aborts than the other processors because of its small transaction capacity, i.e., 8KB for loads and stores, respectively. L1 cache is one of the biggest in the study, with 64 KB is only surpassed by zEC12’s 96KB. L2 cache size is 512 KB. Results show that the suspend and resume instructions are beneficial for avoiding data conflicts on a shared variable to implement ordered transactions.

Diegues et. al [19] and Goel et. al [22] experimentally evaluated Intel TSX implementation on a variety of workloads produced by different benchmarks, more specifically the workloads produced by STAMP benchmark suite [32], microbenchmark Eigenbench [28] and a transactional version of Memcached [27], as well as workloads produced by concurrent data structures. Authors state that TSX has outstanding performance for workloads characterised by small transactions, such as the ones produced
<table>
<thead>
<tr>
<th>Processor</th>
<th>Transactional-store capacity</th>
<th>Transactional-load capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Blue Gene/Q</td>
<td>20 MB (1.25 MB per core)</td>
<td>20 MB (1.25 MB per core)</td>
</tr>
<tr>
<td>IBM zEC12</td>
<td>1 MB</td>
<td>8 KB</td>
</tr>
<tr>
<td>Intel Haswell</td>
<td>4 MB</td>
<td>22 KB</td>
</tr>
<tr>
<td>IBM Power8</td>
<td>8 KB</td>
<td>8 KB</td>
</tr>
</tbody>
</table>

Table 2.1: Transactional store and load of the state of the art processors that successful implement HTM by concurrent data structures and Memcached [27], but only with two of the STAMP benchmarks, namely Kmeans and SSCA.

Goel. et al [22] also compared the performance and energy consumption of TSX with TinySTM [21]. The results shows that TSX performance relies heavily on the access patterns to L1 cache, as every memory access performed inside a transaction is tracked, implying that transactions with bigger read and write-set are prone to exceeding the capacity of the L1 cache and become vulnerable to capacity exceptions and spurious aborts than transactions with read and write-sets that does not exceed L1 cache’s capacity. For low levels of contention, TSX has better performance than TinySTM since TSX does not incur any extra instrumentation and relies upon the cache coherence mechanism of hardware and not in a software library, which by itself incurs an extra cost because every operation must be instrumented in order to track the read and write-set. However as the contention and the size of the read and write-set of the workload increases, TinySTM begins to outperform TSX, as TSX deals worse with contention than TinySTM due to the detection of conflicts at granularity of cache cache line (bytes). TSX scales well up to four threads, however at eight threads, the L1 cache is shared between two threads running on the same core. This cache sharing degrades performance for the larger working set more than for the smaller working set because hyper-threading effectively halves the write-set capacity of TSX. In contrast, TinySTM scales well up to eight threads. In terms of energy efficiency, TSX proves to be more energy-efficient than either TinySTM or the sequential runs.

2.3.1 Lazy Subscription

To ensure that a critical section executed by a hardware transaction does not observe partial effects of a critical section executed by another thread that acquires the lock, transactions subscribe the lock, by reading it and confirming if it is available. Subscribing the lock makes hardware transactions vulnerable to abort if another thread acquires the lock. Typically, transactions subscribe the lock at the beginning of the critical section and are thus vulnerable to such abort during the entire execution of the critical section. Some papers [12; 30] proposed the, so called, lazy subscription optimisation, which delays lock subscription, in order to reduce the duration of this vulnerability.

Dice et. al in this study [16], reported a number of issues associated with lazy subscription. Lazy subscription can cause a transaction to deviate from the behaviour allowed by the original program, as it can result in the thread's registers containing values that could not occur in an execution of the original program. This could result in an access by a transaction to memory that could never be accessed in the original program. If transaction commits, it results in a observably incorrect behaviour.

Compiler support suggested by other publications [10] for avoiding such issues in single global lock-based transaction system is not sufficient to avoid all the pitfalls associated with lazy subscription.
Authors argue that the complexity required to address these issues via static analysis is unlikely to be worthwhile, as there are numerous pitfalls associated with lazy subscription, so manual confirmation of its safety in specific cases is likely to be error prone.

### 2.4 Hybrid Transactional Memory

The usage of a single global lock as a fallback mechanism in best-effort HTM motivated the researchers of TM to create a more complex fallback mechanism, called Hybrid Transactional Memory (Hybrid TM). The goal of Hybrid TM is to exploit best-effort HTM whenever possible due to its cheaper capabilities, and fallback to the more costly STM when a transaction can not complete in hardware. This approach promises to scale well and incur low latency, with the worst-case overhead and scalability comparable to the underlying STM.

To perform as a coherent whole, the Hybrid TM system must be able to detect conflicts arising among concurrent hardware and software transactions. This requirement is usually achieved by logging additional metadata while executing hardware transactions, so that the conflict detection system has access to both sides information. Certain STM algorithms [20; 21] required interaction with per-location metadata, and hybrid versions of these algorithms [37] wasted limited hardware capacity on these metadata. The interaction of HTM with STM metadata could lead to capacity aborts, or could lead to false sharing of cache lines that hold the metadata, resulting in additional HTM aborts, and increased fallback to the STM path. Table 2.2 summarises the key characteristics of state of the art STMs, HTMs, Hybrid TMs and compares them with the two proposed solution in Chapter 3.

#### 2.4.1 Hybrid NOrec

Dalessandro et al. [12] used NOrec STM [13] (see section 2.1.3) as a fallback STM of their Hybrid NOrec to avoid per-access overheads. In this algorithm, the commits of update transactions are executed sequentially, which means that only one transaction can commit at a time. A shared global clock, seqlock, is used to notify concurrent transactions about the updates of both hardware and software.

The original Hybrid NOrec has each transaction start to execute in hardware, and if repeatedly fails to commit, it falls back to executing the NOrec STM in software. To coordinate the execution of hardware and software transactions, three possible integrations between NOrec and HTM are proposed. In naive integration, the same global clock from NOrec, seqlock, is used by hardware transactions to subscribe to software transactions commits notifications. Upon the beginning of hardware transactions, the global clock is read and if is already owned by a software transaction, the hardware transaction spins until the owner completes the writeback and releases the lock. Upon commit, both software and hardware transactions update the global clock to signal their commits to other transactions, which triggers a validation phase in all concurrent transactions. Unfortunately, each hardware transaction conflicts with every software transaction due to the early subscription of the global clock, and aborts when any software transaction commits, regardless of actual data conflicts. Similarly, each hardware transaction conflicts with every other hardware transaction and aborts when any hardware transaction commits.
<table>
<thead>
<tr>
<th>TM implementation</th>
<th>Type</th>
<th>Metadata</th>
<th>HW/SW concurrency</th>
<th>HW capacity used for</th>
<th>Invisible Reads (STM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TinySTM [21]</td>
<td>STM</td>
<td>Per-Region Locks</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>SwissTM [20]</td>
<td>STM</td>
<td>Four-word lock granularity</td>
<td>-</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td>NOrec [13]</td>
<td>STM</td>
<td>Global sequence lock</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>TSX HLE</td>
<td>HTM</td>
<td>No</td>
<td>-</td>
<td>Data</td>
<td>-</td>
</tr>
<tr>
<td>TSX RTM with Global lock</td>
<td>HTM</td>
<td>No</td>
<td>-</td>
<td>Data and Global lock</td>
<td>-</td>
</tr>
<tr>
<td>IBM Power8</td>
<td>HTM</td>
<td>No</td>
<td>-</td>
<td>Data</td>
<td>-</td>
</tr>
<tr>
<td>Hybrid NOrec [12]</td>
<td>Hybrid TM</td>
<td>lock counters</td>
<td>Partial, SW commits abort HW transactions</td>
<td>Data and lock counters</td>
<td>Yes</td>
</tr>
<tr>
<td>Hybrid NOrec-2 [37]</td>
<td>Hybrid TM</td>
<td>Lock counters</td>
<td>Partial, SW commits stall other HW/SW operations</td>
<td>Data and lock counters</td>
<td>Yes</td>
</tr>
<tr>
<td>Hybrid-LSA [37]</td>
<td>Hybrid TM</td>
<td>Per-region lock</td>
<td>Yes, when in different regions</td>
<td>Data and metadata of each region accessed</td>
<td>Yes</td>
</tr>
<tr>
<td>Invyswell [9]</td>
<td>Hybrid TM</td>
<td>Read and Write Bloom-Filter</td>
<td>Depends on the state of the system</td>
<td>Depends on the state of the system</td>
<td>No</td>
</tr>
<tr>
<td>RH NOrec [30]</td>
<td>Hybrid TM</td>
<td>Lock counters and metadata for tracking HTM prefix and HTM postfix</td>
<td>Yes</td>
<td>Data, shared global lock and data from HTM prefix and HTM postfix</td>
<td>Yes</td>
</tr>
<tr>
<td>DMP-SGL</td>
<td>Hybrid TM</td>
<td>Pages read, pages written by STM and transition count</td>
<td>Yes</td>
<td>Data and Global lock</td>
<td>Yes</td>
</tr>
<tr>
<td>DMP-STM</td>
<td>Hybrid TM</td>
<td>Pages read, pages written by STM and transition count</td>
<td>Yes</td>
<td>Data, Global lock and per-thread locks</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2.2: Comparison of various TM state of the art implementations: TinySTM, SwissTM, NOrec, TSX HLE, TSX RTM with global lock, IBM Power8, Hybrid NOrec, Hybrid-LSA, Hybrid NOrec-2, Invyswell, Reduced Hardware NOrec and the two proposed solutions
The second integration approach is called 2-Location, in this approach a second shared location, counter, is added, which decouples subscribing from signalling. Hardware transactions still subscribe to software transactions by reading the global clock but they use counter to signal their own commits to other hardware transactions. The drawback of having a second shared counter is that software transaction must have a snapshot for both the global counter and the second shared counter and must perform additional validation by reading the second shared counter.

The third approach is called P-Location, which differs from 2-Location because hardware transactions no longer conflict with other running hardware transactions, as the counter is now distributed. With this approach, software transactions must poll \( n + 1 \) counters, increasing the overheads. To mitigate that, the number of counters is dynamically determined based on current system conditions.

In order to optimise the algorithm, authors proposed the usage of mechanisms such as lazy subscription, sandboxing and communication filters. In Lazy subscription, the read of the global clock is delayed prior to commit. This increases the concurrency between hardware and software transactions, however it sacrifices opacity [24], as nothing prevents a hardware transaction from reading inconsistent data during software writeback. Authors re-establishes opacity by adding a hardware read barrier that polls the seqlock nontransactionally and pauses if it is locked. Sandboxing is also proposed as an alternative to opacity, rather than instrumenting all loads for consistency, instructions are only instrumented when its effects may expose a transaction inconsistency. With the SW-Exists communication filter, a portion of shared memory is allocated in order to software transactions make their presence visible. When there is no software transaction running, hardware transactions have the possibility of eliding counter updates.

### 2.4.2 Hybrid-LSA

Riegel et al. [37] presented two Hybrid TM algorithms that can execute HTM and STM transactions concurrently and can thus provide good performance over a large spectrum of workloads. The algorithms exploit the concept of speculative (transactional) and nonspeculative accesses, which are non-transactional accesses to memory, i.e., without the addition of memory addresses to transactions' read and write-set. This mechanism avoids conflicts with other running transactions, which decreases the transactions' runtime overhead, abort rate and hardware capacity requirements compared with the versions that use speculative operations.

The paper assumes the availability of nonspeculative operations, which are provided by AMD's Advanced Synchronization Facility (ASF) specification [6], however they are not available in any current HTM implementation, except for IBM Power8, which supports them at a coarser granularity via the suspend/resume mechanism.

The first algorithm, Hybrid-LSA, extends the lazy snapshot algorithm first presented in [35] (overviewed in Section 2.1). LSA relies on the usage of ownership records (orecs) protecting regions of memory and a global time base to check the consistency of transactions. Hybrid-LSA decides at runtime whether to use hardware or software transactions, which both use orecs in order to mediate the access to regions of memory.
The eager variant of the Hybrid-LSA algorithm first performs a load of the oreocz. This operation starts monitoring the oreocz for changes and will lead to an abort if the oreocz is updated. If the oreocz is not locked, the transaction uses a nonspeculative load operation to read the target value, the usage of nonspeculative operation is an efficient way to get the value without adding the address to the read-set.

The write operation is similar to the load operation, but before writing speculatively to memory, the oreocz protecting the memory region is checked for concurrent load and stores (via the PREFETCHW operation, provided by the ASF assembly [6]).

Upon commit, hardware transactions nonspeculatively acquire and increment the clock. This operation sends a synchronisation message to software transactions, notifying them that a hardware transaction is on commit phase and thus, they might have to validate. Next, they speculatively write all updated oreoczs to memory and try to commit. If transactions successfully commit, then is guaranteed that no conflict with other transactions exists.

### 2.4.3 Hybrid NOrec-2

The second algorithm, Hybrid NOrec-2 [37], is an optimisation of the algorithm informally described by Dalessandro et al [13]. The main approach of Hybrid NOrec proposed by [13] is to use a word-sized global sequence lock gsl and an extra sequence lock esl. Software transactions acquire both locks and increment their versions upon commit, whereas hardware transactions monitor esl for changes and increment gsl only on commit. Thus, software transactions are notified about data being concurrently modified by gsl, and use esl to abort concurrent transactions and prevent them from executing during software transactions writeback. The two major problems with this approach is that it does not scale well in practice, as an update of esl will make every hardware transaction to abort, even if both transactions have disjoint working sets. Also, every update of gsl done by a hardware transaction cause every other hardware transactions to abort, even if it is made close to the end of a transaction, as suggested by Dalessandro et al [13].

Authors proposed optimisations to this algorithm using both speculative and nonspeculative operations, which results in a better performance than the original algorithm, with shorter read and write-sets, using simple optimisations.

The first straightforward optimisation consists in having hardware transactions to update gsl only if they will actually update shared state on commit, relieving other hardware transactions from aborting due to the update of gsl. Also, in order to provide more concurrency between hardware transactions that access disjoint data, the write of gsl can be replaced by a nonspeculative atomic fetch-and-increment operation, which allows the algorithm to scale better.

Additionally, hardware transactions do not monitor esl using speculative accesses anymore. The purpose of esl is to prevent hardware transactions from reading inconsistent state such as partial updates by software transactions. To detect such cases and thus still obtain a consistent snapshot, hardware transactions first read the data speculatively and then wait until they observe with nonspeculative loads that esl is not locked. If this succeeds and the transaction reaches the end without being aborted, it is guaranteed that it had a consistent snapshot.
2.4.4 Invyswell

Calciu et al. [9] proposed Invyswell, a Hybrid TM that combines Haswell RTM [19; 22] with software transactions from a heavily modified version of InvalSTM [23]. The main goal of this paper is to improve performance of small to medium-sized transactions that are executed in STM, whose instrumentation cost causes them to perform poorly.

InvalSTM [23] is based on commit-time invalidation. The read and write-sets of a transaction are stored in transaction specific Bloom Filters and upon commit the transaction has complete knowledge of the conflicting running transactions. Authors state that for these reasons, InvalSTM works well with Haswell RTM. RTM is used for short transactions and low thread counts, while InvalSTM is used for large transactions and high thread counts. RTM can leverage InvalSTM use of Bloom filters for conflict detection by augmenting Haswell’s hardware transactions with Bloom filters to enable many hardware transactions to execute concurrently with many software transactions. This enables RTM to perform without interference when read-only software transactions are executing within InvalSTM, regardless of their size.

To manage the shared-memory between RTM and InvalSTM, Invyswell performs the conflict detection after the hardware transaction commits. This is due to the constraints of Haswell RTM, since every write operation done by a transaction is held until commit-time. RTM does not support escape actions, hence when a hardware transaction conflicts with a software one, it aborts. By combining invalidation and conflict detection after a hardware transaction commits, this scheme minimises the chance to abort a hardware transaction due to an in-flight software one.

In order to adapt to different types of workloads, Invyswell support 5 types of different transactions, each one with different characteristics:

- **Speculative Software Transactions (SpecSW)**, which is a type of transactions similar to an InvalSTM transaction, i.e., uses Bloom filters to track the read and write-sets of the memory addresses accessed. The invalidation is done after the commit of hardware transactions.

- **Bloom filter Hardware Transaction (BFHW)**, which uses Bloom filters in order to handle conflicts with SpecSW transactions. After the BFHW transactions commit, it invalidates concurrent SpecSW transactions.

- **LiteHW** transactions, which are lightweight hardware transactions that execute without read or write instrumentation. These transactions can only commit if there are no in-flight software transaction when they begin their commit phase. Because LiteHWs do not maintain read or write-set metadata, if a software transaction is in-flight when a LiteHW enters its commit phase, Invyswell must assume a conflict exists between the LiteHW and the software transaction and, therefore, must abort the LiteHW.

- **IrrevocSW** transactions, which is a software transaction that by being repeatedly aborted, gets a higher priority and acquire a global lock in the beginning of the execution. IrrevocSWs transactions write directly to memory, thus there is no commit phase.
- *SglSW* transactions, which are small software transactions that execute instructions not supported by Haswell RTM, thus need to be executed in software. This transaction type does not use metadata as it writes directly to memory.

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

Inyyswell has been tested with STAMP benchmark [32] and results show that it performs better in a range of benchmarks than NOrec [13], Hybrid NOrec [12] and HLE.

### 2.4.5 Reduced Hardware NOrec

The most recent approach in Hybrid TM is the reduced hardware (RH) proposed by Matveev et. al [29]. The authors applied it to the Hybrid NOrec algorithm, resulting in a new Hybrid TM that overcomes the scalability limitations of Hybrid NOrec [12].

In order to eliminate Hybrid NOrec’s scalability problems, two hardware transactions are added to the software transactions resulting in a *mixed-slow path*. Both of these bottlenecks are caused by reading of the shared clock too early by software and hardware transactions. The first hardware transaction added to the *mixed-slow path*, called *HTM postfix*, encapsulates all the slow path writes at commit point and executes them all together. This change to the *mixed-slow path* enables the hardware transactions to delay the read of the clock to just before committing, avoiding the frequent false-aborts of the original Hybrid NOrec. The other hardware transaction added to the *mixed-slow path*, called *HTM prefix*, executes the largest possible prefix of slow-path reads in a hardware transaction, this is done by starting the *mixed-slow path* as a hardware transaction and executing within it as much reads as possible, or until the first write is encountered. This hardware prefix allows deferring the read of the global clock to after the reads, which significantly reduces the chances of aborting.

Authors state that the algorithm preserves opacity and privatization, as the original Hybrid NOrec. Finally if the *mixed-slow path* fails to commit, the algorithm reverts to the original Hybrid NOrec.

The results of Reduced Hardware NOrec with STAMP benchmark suite [32] indicate that RH NOrec is able to reduce the number of HTM conflicts comparing to Hybrid NOrec [12].

### 2.5 Benchmarks for TM

While several TM systems have been proposed in the literature, there was an urge to develop benchmarks that can correct analyse and compare the proposals. Most TM systems have been evaluated using microbenchmarks, such as linkedlists or red-black trees, which may not be representative of any real-world behaviour, or individual applications, which do not stress a wide range of execution scenarios. Consequently, it has been argued that non-trivial, or realistic, benchmarks are needed to further TM research and to present the "real" benefits of TM.
Some desirable features of non-trivial TM benchmarks are the existence of large amounts of parallelism, executed by code difficult to parallelize using locks, which makes TM more attractive than fine-grain locking. The existence of real-world applications are also desirable because it gives confidence to programmers to use it. Several types of workloads can stress the TM implementations, from long to short transactions, high to low level of contention and different sizes of read and write-set.

2.5.1 STMBench7

Guerraoui et al proposed STMBench7 [25], a benchmark for evaluating STM implementations. The underlying data structure of STMBench7 consists of a set of graphs and indexes intended to be suggestive of more complex applications like CAD, CAM and CASE. A collection of operations is supported to model a wide range of workloads and concurrency patterns. There are long traversals, which go through all the assemblies and/or all atomic parts and update some of them. Second, there are short traversals that traverse via a randomly chosen path. Third, there are short operations, which randomly chose some object (or a few objects) in the structure and perform an operation on the object or its local neighbourhood. Finally, there are structure modification operations, which are operations that create or delete elements of the structure or links between elements randomly.

STMBench7 is likely to benefit STM over HTM and Hybrid TM, as transactions will likely have larger read and write-sets to keep track of objects visited, thus causing capacity aborts on HTM. STM is expected to perform better as the size of read and write-set does not degrade the performance of it.

2.5.2 Lee-TM

Ansari et al. proposed Lee-TM [8], a benchmark based on Lee’s routing algorithm, which consists in circuit routing, the process of automatically producing an interconnection between electronic components. This is achieved based on two phases of the algorithm, expansion and backtracking. Lee-TM has five implementations of Lee’s routing algorithm, which generates a very heterogeneous workload encompassing a wide range of transactions’ duration and length. The benchmark starts by routing the shortest junction in the circuit, generating transactions whose local processing lasts just a few milliseconds. HTM is highly probable to have better performance in this first phase than STM, as transactions are expected to have a small read and write-set (see table 2.1), which lead to overheads due to instrumentation in STM that are avoided by the pure hardware execution of HTM. The benchmark progressively lays junctions of increasing length, generating workloads whose local processing lasts up to a few seconds, which in turn is likely benefit STM over HTM, due to the capacity aborts of HTM incurred by the size of the read and write-set.

2.5.3 STAMP

Minh et al. proposed STAMP [32], a benchmark suite, which enables the analysis of a wide range of TM systems through the use of a wide range of transactional characteristics such as transaction lengths and sizes of read and write-sets.
STAMP consists in eight applications with 30 different sets of configurations and input data that recreate applications from diverse domains, such as e-commerce, Delaunay triangulation, graph processing and circuit routing. The workloads produced by the benchmarks highly variate in terms of time spent inside of transactions, size of read and write-set produced and contention among transactions.

STAMP is portable across different TM implementations, i.e., HTM, STM and Hybrid TM. However, STAMP generally does not work as expected on existing HTM, as it causes nonessential transaction aborts in some of the programs, which motivated researchers to propose a modified version that can fairly compare the intrinsic performance amongst HTM systems [33].

2.5.4 Memcached

Memcached [38] is a distributed in-memory software cache, where multiple clients place key-value pairs on multiple servers. Memcached uses an in-memory hash table that stores key-value pairs. It provides operations such as get, replace, delete and compare-and-swap on these keys. Clients can also request the current status cache, and related statistics.

Tests made by Holla et al. [27] on a transactional version of Memcached indicate that cache locks and stats have high contention but not the data, and propose the use of lock elision [17; 27]. Authors find that lock elision can provide non-trivial benefits to both power and performance, although not all hardware configurations were beneficial due to the existence of false conflicts.

Memcached’s workload is characterised by small transactions, i.e, transaction which the read and write-set is likely to no exceed the capacity of the processor’s cache, which should benefit Hybrid TM and HTM.

2.5.5 Kyoto Cabinet

Kyoto Cabinet \(^1\) is a benchmark suite of Data Base Management data stores written in C++ and produced by Fal Labs. The in-memory component of the suite, Kyoto CacheDB, splits the database into slots, where each slot is a hash table of binary search trees. Each key is hashed into a slot and then hashed again into a hash table of the slot. Kyoto CacheDB first fills the database to a fixed initial size, and then executes the following operations in it with random keys: gets, puts. This benchmark will be likely to benefit STM and Hybrid TM, as with the available operations (gets, puts and deletes) it will search through the index and increase the length of the transaction, and also the size of read and write-set, which consequently cause HTM to abort.

\(^1\)kyoto-cabinet
Dynamic Partitioning

Hybrid TM

The analysis of the state of the art conducted in chapter 2 highlighted that Hybrid TM, although a promising concept, still incurs an overly large overhead, due to the need of synchronization of concurrent transactions using HTM and STM.

The approach proposed in this dissertation is inspired by recent findings [36], which verified that, in several reference TM benchmarks, namely Genome, Vacation and K-Means of STAMP benchmark [32], the memory layout can be partitioned in a way that it is safe to use different concurrency control mechanisms, optimized for different workloads, on different partitions. Nonetheless, static partitioning, i.e., based on static analysis have inherent accuracy limitations, especially when applied to a complex data structures based on pointers. The result is that identified partitions are often overly approximated, i.e., more coarse-grained than they actually need to be, which limits the applicability of using an uncoordinated concurrency control techniques in different partitions.

Based on the assumption that a non-negligible set of the reference TM benchmarks present partitionability in the memory accesses, this thesis proposes a new class of Hybrid TM that takes advantage of the "partitionability" of workloads by minimizing the overheads due to synchronization the between HTM and STM.

Dynamic Memory Partitioning (DMP-TM) avoids to trace conflicts by forcing transactions to log additional information during their execution, but rather to delegate to the OS the function of detecting conflicts, which is done at page-level.

DMP-TM manipulates the process' virtual space in such a way that the heap is mapped twice; one heap is used by transactions running STM and the another one is used by transactions HTM, this way is possible to either one of the TM implementations to revoke the access to the other by simply changing the protection of the corresponding page in its heap. At any point in time, DMP-TM ensures, via OS memory protection mechanism, that each memory page:

- can only be updated by either HTM or STM
- can be shared in read-only mode by both TM implementations

This design was motivated by the goal of sparing HTM from any instrumentation overhead in case it accesses memory regions that belong to its own areas. In case HTM accesses a memory region that does not belong to its own areas, we detect conflicts using the signal handling of Unix and whenever a signal is detected we change the memory protection according to the access being done. This is fundamental to ensure that performance of HTM is not hampered, since, as already discussed, HTM's performance is very sensitive to instrumentation, requiring additional memory accesses given its limited capacity.
This chapter is organized in the following manner: first the algorithm is described in a high level manner, second it is described in a detailed manner. Finally, it is presented the pseudocode that represents the proposed solution and the correctness argument that support this concept.

### 3.1 High Level Overview

#### 3.1.1 Memory Protection

The analysis of the state of art of Hybrid TM conducted in the chapter 2 and the study carried by Diegues et al. [19] reached the conclusion that the state of the art Hybrid TMs implementations, although at first a promising concept, suffer from an overly large overhead caused by the manipulation of metadata in order to synchronize concurrent transactions running in software and in hardware. Hybrid TM implementations rely on fine grained metadata (e.g., per-object ownership records), whose manipulation can inflate significantly the number of memory accesses performed by HTM transactions, and consequently reduce their effective capacity. To avoid this issue, solutions have been proposed that use a small number of, so called sequence locks, which are read by HTM and increased whenever a transaction is committed either in hardware or in software. This requires just a small additional number of reads on the HTM-side, but can cause spurious aborts, i.e., a HTM transaction is aborted whenever a concurrent STM transaction commits, even if they had not developed any actual data conflicts.

This thesis proposes a novel synchronization mechanism which aims to overcome the aforementioned limitations of state of the art Hybrid TM by exploiting a property that was recently verified in several reference TM benchmarks [21], which we refer to as partitionability: the memory layout and transactions of these applications can sometimes be partitioned such in a way that no two transactions belonging to different partitions will ever conflict on a common memory region.

The partitionability present in the workload is either spontaneous, being already present in the workload, or by using conflict-aware partitioning techniques, such as the ones presented by Riegel et al. [36]. This solution uses static partitioning at compile-time to construct a Data Structure graph, and subsequently go through all the pointers to check the limit where they are permitted to point, which defines the partition.

Static partitioning has inherent accuracy limitations, especially when applied to complex, pointer-based data structures. As a result, the partitions that static partitioning identifies are often overly approximated, i.e., more coarse-grained than they actually need to be. This limits the applicability of the idea of using uncoordinated concurrency control techniques in different partitions. In light of this analysis, DMP-TM uses dynamic partitioning in order to adapt the partitions at runtime and achieve better throughput than a static approach.

However not a primary goal, we wanted to support workloads that are not perfectly partitionable. Therefore we needed to have a synchronization mechanism to detect conflicts between transactions running in HTM and transactions running in STM. Thus, we rely in the memory protection mechanism of hardware, a mechanism used by processors and operative systems to prevent processes from accessing certain memory addresses, to detect conflicts between HTM and STM. In modern operative systems,
this protection is enforced by using the system call \texttt{mprotect()}, and we use it to restrict the access of HTM to a certain page and minimize the cost of detecting possible access violations, which may lead to conflicts with concurrent STM transactions. There are 3 possible types of HTM data protection modes:

- Read/Write protection - In this protection mode, HTM can read and/or write any data present in the page.
- Read protection - This protection mode allows STM and HTM to run in parallel as long as none of both issues writes, as two concurrent reads to the same object does not represent any conflict.
- None protection - In this protection mode, HTM is not allowed to Read nor Write any address of the page.

Whenever HTM accesses a memory address which the page protection is not compatible with the type of access done, it is triggered an Access Violation (AV), for example if HTM issues a read when the protection is None there it will be raised an Access Violation, however if HTM issues a write when the page protection is Read/Write, then there won’t be triggered any AV. The AV is caught by a \texttt{signal handler} that calls the Contention Manager module. The Contention Manager’s functionality is to decide what is done in order to fix the AV and retrieve this information to the \texttt{signal handler}, that will do the necessary changes.

### 3.1.2 Mapping of Memory

In order to allow STM and HTM to have different access rights to the same physical memory page, we map each physical page to two different virtual addresses, one to be used by HTM transactions and one by STM transactions.

The result is if a page’s access rights is revoked, there is not a way to distinguish accesses performed by transactions running STM, which are always enabled by this concept from transactions running HTM, that must check the page protection first. This happens because the page’s access rights are per-process, which makes pages inaccessible for the entire process, thus for both STM and HTM.

In order to solve this, we organize the process virtual space in a way that the heap is mapped twice (figure 3.1): transactions running HTM access the corresponding pages in a mapping of the heap reserved for them, called HTM Heap, and in a similar way, transactions running STM access the corresponding pages in a mapping of the memory called STM Heap.

Finally, in order to minimize instrumentation overheads on the HTM side, during HTM’s transaction execution it is not checked at the TM library level whether it has the correct access rights to serve that access. Conversely, DMP-TM relies entirely on the OS/Hardware memory protection mechanism to notify, via a SIGSEGV signal, if a HTM transaction tries to access a page that belongs to the STM partition.

On the STM side, on the other hand, DMP-TM takes a different approach. Before issuing an access, it is first verified if the page is accessible and, in the negative case, it delegates to an external module, which we call Contention Manager, how to manage the scenario, i.e.:
assign the access right to STM. This can in theory be done immediately - which is the solution adopted in DMP-TM, or possibly after forcing a wait aimed to allow concurrent HTM transactions to commit before revoking its access rights.

• abort the STM transaction and restart it in hardware.

Depending on the access performed by STM we change the protection of the heap accordingly: If STM issues a write operation, the protection of the corresponding page on HTM Heap is changed to None (figure 3.5) and similarly if STM issues a Read (figure 3.2), then the protection of the corresponding page on HTM heap changes to Read. The result of this approach is that if transactions running HTM tries to read or write a page for which it does not have the access rights, the operative system checks that the process does not have the access rights to do it and triggers an Access Violation. The Access Violations informs both transactions running HTM and transactions running STM that the other type of transactions is accessing the same partition as them. Otherwise, the transaction running HTM is accessing a page not accessed by STM transactions and the operations are done without any instrumentation or changing the access rights.

Figure 3.1: Map of the address space in the HTM Heap and in the STM Heap

Figure 3.2: Transaction running STM performs a read operation
3.1.3 HTM Component

The HTM implementation used in this solution is the one provided by IBM Power8. The leading cause that motivate us to choose this specific processor instead of the more common processor Intel's Haswell is that in Intel’s Haswell when a transaction running HTM performs an access on a shared memory zone and triggers a Segmentation Fault (SIGSEGV) signal, we have experimentally verified that the signal handler is not executed, further making the system consistently retrying the transaction.

If aborted by a capacity abort, transactions running HTM immediately fallback to one of the fallback mechanisms described more in detail in subsection 3.2.8, i.e., either acquiring the fallback single global lock or run the transaction using STM. In case of a conflict, the transaction is repeated up to 5 times - a standard value that has been recommended in the study of Yoo et. al [42], before activating the fallback path.

3.1.4 STM Component

Motivated by the study carried out by Diegues et. al [19] that compared several TM backends in different workloads, we opted to integrate in DMP-TM TinySTM, a very efficient word-based STM, which has been shown to have very robust performance across heterogeneous workloads.

As mentioned before, we want to avoid instrumentations on HTM, therefore in DMP-TM we opt for placing any additional run-time check aimed to enforce correct synchronization between HTM and STM, on the STM side. This is motivated by the fact that STMs need anyway to perform checks to detect conflicts, which allows amortizing the additional costs associated with detecting access violations to memory partitions. To this end, we associate with each memory page the following metadata: 1) status field, that encompasses the access rights that HTM currently has on that page (read, read-write or none), 2) transition count, that encompasses how many times the access rights for HTM have been restored to read/write, 3) writer count, that encompasses how many STM transactions have written the page.

By storing the former metadata at the application level, DMP-TM can spare expensive system calls to check what is the status of a page. The latter metadata is used instead to determine whether some of the pages previously accessed by a STM transaction may have been concurrently accessed by a HTM transaction - a scenario that DMP-TM handles by aborting the STM transaction (see figure 3.4.

3.1.5 Signal Handler and Contention Manager

When a transaction running HTM tries to access data from a page to which it is not allowed, the operative system raises a SIGSEGV signal. This signal is handled by a Unix handler, where is possible to check the address that originated the SIGSEGV and consequently calculate the page that triggered it.

Then, the signal handler passes the information of the ID of the page that caused the conflict as well the type of the transaction that caused the conflict to the Contention Manager module. The information
of the type of a transaction, i.e., if a transaction is read-only or if a transaction is an update transaction is specifically passed by the programmer.

Then, the Contention Manager replies to the signal handler with an action to be applied. Currently the Contention Manager supports the following replies:

- **CHANGE_PROTECTION_READ/WRITE** - By confirming that the type of the transaction that caused the conflict was update, the Contention Manager advises changing the protection to Read/Write.
- **CHANGE_PROTECTION_READ** - By confirming that the transaction that caused the conflict was read-only, the Contention Manager advises to change the protection to Read.
- **BATCH_SYSTEM_CALL** - This reply denies the change of protection to the page that received an AV and forces the transaction to use the fallback mechanism. This reply is intended to make DMP-TM batch the system calls and avoid issuing constantly system calls to change the protection by restoring the page protection after a predetermined batch of AVs.

One can envision a number of possible approaches to implementing the Contention Manager logic, e.g., comparing the costs and benefits of the change in the access right to the page using various metrics, white-box or black-box models and historical data. A key aspect here is to ensure that the decision is taken efficiently, as the computational costs of the Contention Manager lie on the critical path of execution of transactions.

Motivated by this consideration, DMP-TM currently integrates a simple, and hence very lightweight, heuristics aimed at avoiding an unbounded number of ping-pong of a page between the STM and HTM memory partitions: the migration request is denied as soon as the transition count reaches a predetermined threshold, and makes DMP-TM use the fallback mechanism, namely falling back to a single global lock (SGL) and falling back to plain STM that does not use memory mechanisms. In order to ensure that these mechanisms are safe to use, the former implies blocking all STM transactions, given that HTM transactions are already blocked whenever the SGL is activated, as discussed in Section 2.3. The latter fallback mechanism, instead, blocks all HTM transactions, but allows for parallelism with STM transactions, as these use the same concurrency control algorithm.

In contrast to the typical Hybrid TM implementations, where every transaction start running in HTM and then fallback to STM if HTM cannot complete, DMP-TM uses the Scheduler Contention Manager module to assign the partitions via offline static analysis. This module can be further improved to grant online hints about which type one partition should have. For instance, one approach would be for the Scheduler Contention Manager to learn that a partition is suitable for STM if HTM transactions consistently get capacity aborts in it.
3.2 Detailed description

3.2.1 API

This section is devoted to presenting the API exposed by DMP-TM. The set of functions, whose signature and semantics are described in the following, are meant to be used either directly by the programmer (e.g., to demarcate transactions) or automatically injected at compile time (e.g., HTM_READ() or STM_READ could be used by a compiler to generate the HTM or STM code path, respectively). Recall that DMP-TM has not been integrated in a compiler (like GCC), yet. Hence, currently these APIs need to be explicitly used by the programmer during the development of their applications.

- **TM_STARTUP (integer numThread)** - Must be placed in the beginning of the program, it starts all the metadata, maps the memory in two heaps, calculates the offset between heaps, calculates the range of addresses available for each thread to allocate memory and initializes TinySTM.

- **TM_SHUTDOWN ()** - Must be placed at the end of the program, it ends TinySTM and calculates all the statistics, e.g. throughput, number of aborts, number of commits, etc. It also presents this information to the user.

- **TM_THREAD_ENTER ()** - Must be called in the beginning of a thread so that TinySTM must be aware of its operations.

- **TM_THREAD_EXIT ()** - Must be called when the thread finishes its execution.

- **TM_MALLOC(integer size)** - This directive is placed whenever the programmer wants to allocate a bucket of size elements.

- **TM_BEGIN_EXT (integer mode, boolean ro)** - this directive is used to start a new transaction, either using HTM or STM. The variable mode indicates the type of transaction created, if equals to 0, it will begin a transaction running HTM, otherwise it will begin a transaction running STM. The ro parameter indicates if the created transaction is read-only or not.

- **TM_END ()** - this directive tries to commit the transaction created by TM_BEGIN_EXT (b,mode,ro).

- **HTM_READ (integer var)** - issues a read of var inside of a transaction running HTM.

- **HTM_WRITE (integer var, integer val)** - writes val to var inside of a transaction running HTM.

- **STM_READ (integer var)** - issues a read of var inside of a transaction running STM.

- **STM_WRITE (integer var,Integer val)** - writes val to var inside of a transaction running STM.

3.2.2 Operating system mechanisms used for memory mapping and protection

In order to solve the problem of having to revoke the access to certain pages for the transactions running HTM, but not for the transactions running STM, we first created a shared memory zone using the directive `shm_open()`. The mapping must point to the same object in memory, otherwise the two
mappings would point to two different memory regions and one update in one region would not be reflected in the other region. Then, we used Unix system call *mmap()* to organize the process’ virtual space in such a way that the heap is mapped twice; one heap, named HTM Heap, is used by transactions running HTM and the another one, STM Heap, is used by STM transactions. This design decision allows to revoke the memory accesses to certain pages whenever they are accessed via HTM Heap but not whenever they are accessed via STM Heap.

Because accesses performed by STM transactions must revoke the access from the corresponding page in the HTM Heap, the conversion of addresses between one heap and the other must be performed in the most efficient way possible. For that reason, the two heaps are placed at a constant offset one from another, and this offset is calculated at runtime. This way, in order to know the corresponding address in the HTM Heap, we sum the address in HTM Heap with the offset between heaps. This implies basically two important changes in the way applications should be developed: 1) memory allocation, which should be done using only memory pages in the shared memory region. This can be automated, and be made fully transparent to applications, using specialized implementations of malloc - which is the solution adopted in this thesis. 2) two code paths need to be produced, one for STM and one for HTM transactions, which target the correct range of shared memory addresses. Given that the translation between the two address spaces is simple (just a plain translation) and this task can be automated by a compiler. However, the current version of DMP-TM does not provide compiler support for this task, hence requiring programmers to explicitly generate a second code path with translated memory addresses.

The revoke of access rights is achieved by using the system call *mprotect()* of Unix. This system call changes the protection of calling process’ memory pages contained in a given range of addresses. *Mprotect()* must be issued on a piece of memory obtained using *mmap()* and the given address must be aligned at the page boundary, which in DMP-TM is, otherwise, according to the POSIX standard, its behavior is undefined.

The result of this concept is that whenever transactions running HTM try to access a page for which they do not have the access, the operative system detects that the process does not have the access rights to do it and triggers an AV in the form of a SIGSEGV. This signal is further treated by a Unix *signal handler*.

### 3.2.3 Synchronization of HTM and STM component

#### 3.2.3.1 Page Metadata

In order for the STM transactions to detect that HTM has changed the protection of a page, thus possibly overwriting a position of memory previously read by STM transactions and causing a conflict, we used a per-page structure to store the metadata of each page. The per-page metadata encompasses the following information:

- **Status field** - Encompasses the information of which page protection does the corresponding page in the HTM Heap has in the moment. Because of that, it avoids issuing an extra system call to check which is the current protection of a certain page in the HTM Heap.
- **Transition Count** - Transactions running in software read this variable upon their first access to a page. Whenever they issue a subsequent access to the same page, and before committing, STM transactions check this variable again, in order to detect whether, in the meanwhile, some HTM transaction has restored the access rights in an incompatible mode. This indicates that a conflict may have arisen between the active STM transaction and some concurrent HTM transactions. This variable is incremented in the signal handler whenever an AV is triggered and the page protection is restored to Read/Write mode. This variable is always incremented in an atomic way recurring to the function `__sync_add_and_fetch()`, this way other threads do not see intermediate values. Every transaction has a variable signaled by the programmer, `ro`, that indicates if a transaction is read-only or not, in the case that the transaction that wants to restore the protection is an update transaction, then the transition count is incremented. Otherwise, the protection is restored and the transition count is not incremented.

- **Writer count** - This variable is atomically incremented whenever update transactions running STM are active. Transactions running STM are only alerted by HTM transactions that have restored the protection to Read/Write mode. This leaves opened the case that STM transactions continue to write to memory and transactions running HTM, that have restored the protection to read mode, read wrong values and break correctness. This way, and in order to not break correctness, HTM transactions that want to restore the protection wait until the writer count is equal to 0.

- **Lock Bit** - Because if two different transactions issue two `mprotect()` calls to the same page, its behavior is not defined, we use a variable called `lock bit` in order to guarantee that at most one transaction can change the protection of a certain page and subsequently the status field of the page. We use the directive `compare-and-swap` to check for the value of the variable; if it is not held by other transaction, we swap the value atomically and avoid, because of the parallelism of multiple threads, that the lock bit end up being held by more than one transaction.

In addition, each transaction maintains the information of the transition count of each page read, in the form of an array indexed by the page identifier, and whose entries store the transition count observed by the transaction for a given page. This information is used to check if the set of pages read is still consistent, i.e., if the transition count has not changed meanwhile.

### 3.2.4 Signal Handler

As stated before, transactions running HTM detect conflicts by getting an AV whenever they access a page for which they do not own the necessary access rights (see algorithm 1). To implement this behavior, at the beginning of the program is registered a signal handler that intercepts SIGSEGV signals and treats them. In the case of an access violation, it calls the Contention Manager component in order to determine which action must be performed in order to solve the conflict. If the Contention Manager grants permission to change the page protection, the signal handler tries to grab the `lock bit` using the directive `compare-and-swap`. This way is guaranteed that the transaction is the only one changing the protection of the page. Further, it checks the number of STM writers active in the page. If there are writers active in the page, it releases the `lock bit` and waits until there are not any writer active. If there are not any STM writer active at the page, it changes the protection of the page to the one given by the
Figure 3.3: Scenario where STM and HTM transactions abort each other

Contestion Manager module using the `mprotect()` directive, changes the Status field to the protection recently changed. Finally in the case of changing the protection to Read/Write, the signal handler atomically increases the transition count of the page and also the number of total transition counts, an optimization described in subsection 3.2.7.

3.2.5 Contention Manager

The Contention Manager has the responsibility of stating which protection of the page must be restored whenever an AV is raised, it analyses the state of the system after being called by the AV Handler (see algorithm 2).

Currently, the contention manager makes the decision based on two different parameters: the type of the transaction that caused the AV and the number of times that a page has its protection restored to Read/Write. If the

In Figure 3.3 is presented a scenario where an STM transaction issues a read and revokes the access of HTM to a certain page. Then, HTM tries to issue a transactional write and gets an access violation, for which the Contention Manager restores the protection and further increases the transition count. By the time the STM transaction issues the next read or tries to commit, it gets aborted as the transition count has already changed. The STM transaction aborts and restarts, issuing again a read and revokes the access for HTM transactions. If the HTM transaction as not yet committed, it will abort because the updates of memory are all done at commit time, and the protection of the page written does not allow to write at the commit-time, which makes the HTM transaction to abort.
To avoid entering in a loop where each type of transactions abort the other one, the Oracle first checks how many times the page protection has been restored. If the number of times that the protection has been restored is greater than a predetermined threshold, then the system does not restore the protection of the page. Instead, it uses one of the fallback variants described in section 3.2.8 to conclude the task. If the number of times that the page protection has been changed is lower than the predetermined threshold, then the system checks the type of the transaction that caused the AV and chooses one of the following behaviors:

- If the transaction that caused the AV is read-only, then the system will change the page protection to Read, this allows the STM and HTM component to run in parallel has two reads to the same address does not represent any conflict.

- If the transaction that caused the AV is an update transaction, then the system will change the page protection to Read/Write in order to have rights to write.

3.2.6 STM Algorithm

3.2.6.1 Reads performed by STM transactions

When a transaction running STM issues a read (see algorithm 6), it first checks if the transition count of all the pages previously read have been changed since the last time they were read (line 3 of algorithm 6). If any of the transition count stored is different than the present one, then the transaction is restarted (line 6 of algorithm 6) since an HTM transaction has restored the memory protection of the page and possibly issue a write to an address previously read by the STM transaction (see figure 3.4).

Then, it is checked the metadata of the page is to which the transaction running STM issued a read. If the protection of the corresponding page in the HTM Heap is Read/Write, which means that HTM transactions can write a value whenever STM is reading, then the lock bit is grabbed using the directive compare-and-swap, which guarantees that only one transaction acquires the lock bit and issues a system call mprotect in order to change the protection of the page to Read (line 15 of figure 3.2).

We considered which change of page protection should a read done by STM trigger and it is preferable that instead of revoking, we opted for preserving read rights for HTM transactions. This choice allows in fact concurrency between STM and HTM transactions that read, without updating, memory positions in that page. After altering the metadata of the page, it is issued a transactional read recurring to the function stm_load() of TinySTM and it is checked again if the transition count has changed. If it does, it means that the performed read is not legal and consequently the transaction is restarted. If not, the read has been successful.

3.2.6.2 Writes performed by STM transactions

When a transaction running STM issues a write operation (see algorithm 7), it first checks the page’s metadata. Then, it checks the status field of the page containing the address and if it finds that the page that wants to be written is readable or writable by transactions running HTM, it acquires the
lock bit using the compare-and-swap directive and revokes the access rights. This is done by issuing a `mprotect()` call and changing the protection of the corresponding page in the HTM Heap to None, so that no HTM transaction can read it or write it in middle of an STM write. After changing of the protection via `mprotect()` to None and furthermore update the status field of the page, the writer count for that page is atomically increased (see line 8 of algorithm 7). Finally, it is released the lock bit and is performed the transactional write of the value recurring to the function `stm_store()` of TinySTM (see line 11 of algorithm 7 and figure 3.5).

### 3.2.6.3 Restart of an STM transaction

Whenever a software transaction finds the transition count of a page different than the one read by the time the read is done, the software transaction begins the process to restart itself: First, recurring to the function `__sync_sub_and_fetch()`, it atomically decreases the number of writers for all the pages that the transaction has written before and finally issues a transactional restart recurring to the function `stm_abort()` provided by TinySTM.
3.2.6.4 Commits performed by transactions running STM

By the time a Software transaction finishes its execution, it enters in the commit phase (see algorithm 8), in which it first checks if the transition count of the pages previously read have changed meanwhile. In the case that they have changed, the transaction restarts as explained in subsection 3.2.6.3. In the case that no transition has been incremented, the transaction commits recurring to the function $stm\_commit()$ provided by TinySTM.

3.2.7 Optimizations

This section presents two optimizations integrated in DMP-TM in order to maximize its efficiency. First, we present a bitmap used to accelerate the process of finding the pages read and written by a STM transaction and finally, we introduce a global transition count that aims to skip the process of validating all the transition count, while assuring correctness.

3.2.7.1 Bitmap representing pages read and written

DMP-TM uses two bitmaps to optimize the process of checking which pages have been read/written. For the cases where DMP-TM checks if a certain page has already been read or written, DMP-TM uses an array indexed by page where each index represent: 1, a page has been read/written; 0, a page has not been read/written.

Also, it uses an additional array in conjunction with a counter of pages read intended to avoid overheads in the process of revising if the transition count of the pages read has changed. By using this array together with the number of pages read is possible to read precisely the number of pages read and avoid having to go through all the elements of the array.
3.2.7.2 Usage of a Global transition count

To ensure correctness, STM transactions must check, upon each read, if the transition count of any of the pages they read so far has changed in the meanwhile. Thus, each read incurs an overhead that is linear with the number of pages read so far. In order to tackle this issue, we use a global transition count that is increased atomically whenever a transaction running HTM restores the protection of a page to Read-Write mode.

This way, transactions running STM store the value of the global transition count at the beginning of their execution and each time they issue a read operation, check if the value of this variable has changed in the meanwhile. If it did not, they skip having to check all transition counts because is guaranteed that no transaction running HTM has restored any page protection. Otherwise, they store the new value of the global transition count locally and check if the transition count of any of the pages read so far has changed.

In a similar way, at commit-time, if an STM transaction checks that the global transition count is equal to the local safe copy of global transition count, it skips checking if the transition count of all the pages read.

3.2.8 Variants of the algorithm

This section describes the variants of the algorithm developed in order to accommodate the cases when the solution reaches the point that it cannot succeed using hardware transactions, either because of reaching the maximum number of tries in hardware or reaching the maximum number of page’s protection changes. Thus having to fallback to other solutions.

The fallback mechanisms can be coarsely divided in two approaches:

- Resorting to executing the transaction sequentially, i.e., without any other transaction executing concurrently. This is the standard fall back approach for HTM. In this case, though, it has to be extended to ensure that neither HTM nor STM transactions are active concurrently to the transaction executing in the fallback path. As we will discuss next, we have developed three variants of this approach, which use different ways to block the execution of STM transactions.

- Stopping the execution of HTM transactions (by acquiring the global lock they subscribe upon begin), and running the transaction using a plain version of TinySTM.

3.2.8.1 Fallback to Single Global Lock

This solution uses the algorithm described before and whenever the maximum number of retries is reached, which we found 5 tries to be the best value, it fallback to running the code that would be executed by a hardware transaction in a non-instrumented way. With this approach, it was expected to benefit workloads characterized by more than 90% of transactions running HTM, since whenever HTM transactions fallback, it does to a non-instrumented version of the code, which is the quickest way to execute a program. However, to assure that non-instrumented code execution causes a conflict with a
running software transaction, it is grabbed a global lock that is also subscribed by HTM transactions, so that whenever there is a non-instrumented code execution, no hardware transaction will be running. Additionally, it is needed to block the execution of all the software transactions in order to avoid possible conflicts with the uninstrumented execution. This feature is achieved by having a per-thread lock that is \textit{signaled} each time a software transaction begins and \textit{de-signaled} when it ends; whenever hardware transactions fallback to using the global lock, they keep trying to acquire all the per-thread locks, which further blocks the execution of software transactions.

3.2.8.2 Fallback to TinySTM

This solution fallback to using a plain TinySTM with no extra-instrumentation on top of it. This is achieved by using a global lock that is grabbed whenever a hardware transaction exhausts its number of tries, which stalls all the other running hardware transactions. Then, it starts executing TinySTM with no extra-instrumentation on top of it. The execution of the fallback transaction is compatible with the execution of the normal software transactions, which uses instrumentation on top of TinySTM in order to keep track of the pages read and written and also in order to change protection for the HTM addresses. We choose to use a non-instrumented version of TinySTM has the fallback mechanism, first because it is faster and second it does not change the memory protection of a partition that is being used by hardware transactions.

3.2.8.3 Falling back to single global lock acquired by hardware transactions

This solution fallback to acquiring a global lock and running uninstrumented. Software transactions signal their commit using a per-thread lock at the beginning of the transaction and de-signal it whenever the transaction ends. Hardware transactions revert to uninstrumented execution by acquiring the global lock and all the per-thread locks used by Software transactions to signal their execution. This approach is similar to the one described before with the difference of the method used to acquire the locks. Before, it was used an atomic \textit{compare-and-swap}, which is an operation that \textit{atomically} checks the value of a variable and if it is equal to a certain value, changes its value to a given value.

In this approach, it is used hardware transactions, that are faster than \textit{compare-and-swap} to acquire the locks. Although, this approach does not guarantee to be successful as multiple hardware transactions trying to acquire the same lock could abort each other, or by exhausting cache capacity, so in order to guarantee progress after retrying 5 times to acquire the lock using HTM transactions, the thread starts to use \textit{atomic compare-and-swap} to do it.

3.2.8.4 Falling back to Global lock using memory barriers

Memory barriers are special types of barrier instructions that cause the central processing unit (CPU) or the compiler to enforce an ordering constraint. Since modern compilers re-order the loads and the stores in such a way that the loads and stores are optimized and the correctness of a single thread execution is not compromised, this is not true for multi-threaded execution, as the re-ordering of the loads and stores lead to a break of correctness.
By using memory barriers, we avoid using atomic operations or hardware transactions, since we can just use loads and stores to one variable. The system works as follows: Every thread that runs STM transactions has one variable that indicates in which state of the system they are presently, there are 3 states:

- **Thread can execute** - In this state the thread executing an STM transaction can execute.
- **Thread has received a stop request** - In this state the thread executing an STM transaction has received the request to stop by an hardware transaction that wants to fallback and will finish its execution and avoid that the thread run any other Software transaction.
- **Thread is stopped** - In this mode the thread cannot start any software transaction because there is uninstrumented code running.

Whenever an HTM transactions wants to fallback to global lock, it first acquires it, making all the HTM transactions abort and subscribe for its release. Then, it sends a signal to all the threads running STM transactions to go to the state "Thread has received a stop request", which make it the last transaction until the uninstrumented execution begins. Then, when STM transaction finishes, they go to the state "Thread is stopped" and they signal the thread that acquired the global lock. When all the threads have signaled the one holding the global lock, then it can execute. When the uninstrumented execution finishes, it sends a signal to all the threads, so that they can go to the “Thread can execute” state.

### 3.2.9 Memory Allocation

Recall that our solution relies on mapping the memory accessible via TM in specific memory regions of the address space. This implies that the mechanism in charge for allocating/deallocating memory dynamically (malloc/free in C) should be made aware of the range of addresses from which memory chunks should be allocated upon request of the application.

In order to address this issue, we developed a custom implementation of malloc that allocates memory from the mapped STM heap. In the beginning of its execution, the STM Heap is split in \( n \) equal parts aligned with the page's boundary, being \( n \) the number of threads. Then, when is needed to allocate a new piece of memory, the allocator returns one address of the split reserved for that thread and increments the per-thread counter that indicates the point to where, in the partition, memory is allocated.

The key benefit of this solution is that the memory allocator does not have to deal with deallocation of memory, as each thread consumes memory from different memory regions. This approach, although simple, comes with a cost: if a thread finishes its split, and even if the other splits are empty, it cannot access those memory regions. However, the development of an efficient malloc support is outside of the scope of this dissertation and that the current, simple approach was sufficient to carry out the evaluation of the system.
3.3  Pseudocode of the solution

This section presents the pseudocode of DMP-TM. First, it presents the pseudocode used by the signal handler, the Contention Manager and the pseudocode for changing the protection for HTM to Read and Read/Write mode. Then, it presents the pseudocode used to start HTM transactions. Finally, it presents the pseudocode used by STM transactions, including reads, writes, commits and restart operations.

3.3.1 Signal Handler Pseudocode

Algorithm 1: Signal Handler

```plaintext
1 Function: Signal Handler(pageID)
2 Reply = Contention Manager(pageID)
3 Change protection to the one given by the Contention Manager (either Read, Read/Write or use the fallback mechanism)
```

3.3.2 Contention Manager Pseudocode

Algorithm 2: Contention Manager

```plaintext
1 Function: Contention Manager(pageID)
2 if transaction was an update transaction then
3     return Change page protection to Read/Write
4 end
5 if transaction was a Read-only transaction then
6     return Change page protection to Read
7 end
8 if Maximum number of page protection change reached then
9     return Use Fallback mechanism
10 end
```
3.3.3 Change of protection to Read Pseudocode

Algorithm 3: Change page protection of HTM to #Read

1 Function: Change_protection_HTM_Read(pageID)
2 while writer_count[pageID]!=0 do
3 | spin
4 end
5 Acquire lock bit that prevent other transactions from issuing system calls
6 if writer_count[pageID]==0 then
7 | if HTM can write the page then
8 | Change Page protection to #Read
9 | Change protection in metadata of page to #Read
10 end
11 Release lock bit
12 end

3.3.4 Change of protection to Write Pseudocode

Algorithm 4: Change page protection of HTM to #Write

1 Function: Change_protection_HTM_Write(pageID)
2 while writer_count[pageID]!=0 do
3 | spin
4 end
5 Acquire lock bit that prevent others from issuing system calls
6 if writer_count[pageID]==0 then
7 | if HTM can read or write the page then
8 | Change Page protection to #Read/Write
9 | Change protection in metadata of page to #Write
10 end
11 Release lock bit
12 end
3.3.5 Begin HTM Pseudocode

Algorithm 5: Begin HTM transaction

1 Function: Begin HTM(readonly)
2 while Global lock is acquired do
3     spin
4 end
5 Begin HTM transaction
6 if HTM transaction aborts then
7     retries := retries-1
8     if Number of retries exhausted then
9         Execute fallback mechanism
10    end
11 end
Algorithm 6: STM Read

Function: STM read(var)
Read global transition count and store it in temporary variable

if global transition count has changed from the first transition count read then
  for All pages read do
    if Transition count of page read has changed then
      Restart Software Transaction
    end
  end
  Store temporary variable in local transition count
end

if first time page is read then
  Add page to set of pages read
  Store transition count of page in array containing the transition count of pages previously read
end

if HTM can write the page then
  Change page protection of HTM to Read
  Change metadata of page to Read
end

Transactional Read
Read Global transition count to a temporary variable

if Local transition count different than temporary variable then
  for pages read do
    if First time page is read && page's transition count has changed then
      Restart Transaction
    end
  end
end

Store value of temporary variable in local transition count
### 3.3.7 STM Write Pseudocode

**Algorithm 7: STM Write**

1. **Function:** STM write(var)
2. **if HTM can read or write the page then**
   3. Acquire the lock bit in order to prevent other transaction issuing system calls
   4. **if HTM can read or write the page then**
   5. Change Page protection to #Read/Write
   6. Change protection in metadata of page to #Read/Write
   7. **end**
   8. Release lock bit
   9. **end**
10. **if first time page is written then**
11. Increment number of writers in this page
12. Add page to Write-set
13. **end**
14. Transactional write

### 3.3.8 End STM Pseudocode

**Algorithm 8: End STM Transaction**

1. **Function:** STM commit(var)
2. **for each page read or written do**
3. **if global transition count has changed then**
4. **if page was read then**
5. **if transition count of the page has changed then**
6. Restart Transaction
7. **end**
8. **end**
9. **end**
10. **if page was written then**
11. Decrement number of writers of page
12. **end**
13. **end**
14. Commit changes
3.3.9 STM Restart Pseudocode

Algorithm 9: STM Abort

1 Function: STM abort(var)
2 for for each written page do
3    if page was previously written then
4      Decrement writer count
5    end
6 end
7 Clear set of pages written
8 Clear set of pages read
9 Abort Transaction

3.4 Correctness Argument

In this section, we provide a set of (informal) arguments on the correctness of the proposed solution.

As for HTM transactions, they are effectively isolated by concurrent updates by STM transactions as a HTM transaction is aborted as soon as it attempts to access a memory address for which it does not hold adequate (Read or Read/Write) permissions.

The cases where HTM transactions read a value from a page and, prior to their commit, a STM transaction begins and writes to the same variable are straightforward: Because cache coherence mechanism uses physical addresses to detect conflicts, it ensures correctness even when the accesses are issued on different virtual addresses that point to the same physical address.

The case where pages have been written by software transactions are straightforward: each transaction increments the writer count atomically and revokes the access for HTM transactions before issuing the transactional write. On the signal handler side, if the transaction wants to revert to Read/Write protection, the AV handler waits until there are no writers active and then acquires the lock bit atomically, which guarantees that it is the only transaction that can execute system calls on that page.

The cases where pages have been read are handled in the following manner: Before STM's first read to a page, it has to store the page's transition count and also check if the access protection for transactions running HTM is either None or Read-only. In the negative case, it acquires lock bit and issues a system call in order to change that protection to Read. After it has performed the transactional read, the transaction must check that the transition count read before remained unchanged. This means that the page was never written by HTM, because if it has changed the protection meanwhile, the transition count would be different than the previous one.

Finally, at commit time, all the transition counts of the pages read are checked, otherwise some HTM transaction could write meanwhile and invalidate the values read by STM transactions before. In order to ensure opacity,

the transition count of the pages read, so far, by an active STM transaction are also checked. This
ensures that no HTM can have conflicted with the STM transaction, by overwriting any of the values it read so far.
This chapter presents an extensive experimental evaluation of DMP-TM. In order to stress different scenarios, we used a testbed composed by microbenchmarks with 300 different workloads plus real life applications, such as Genome, Intruder and Vacation from STAMP [32]. The results of the experiments were compared with other state of the art of the art TM implementations, namely HTM falling back to Single global lock (HTM-SGL), TinySTM [21], NOrec [13] and Hybrid-NOrec [12].

This extensive study lead us to track the workloads were DMP-TM is advantageous in comparison with the state of the art of TM but also workloads where DMP-TM is not advantageous and delimit where is the landmark where this solution starts to having performance penalties.

4.1 Methodology and Evaluation Metrics

First, an evaluation using synthetic benchmarks is presented, i.e., microbenchmarks created by the authors, that are adjusted in order to generate diverse workloads with three key factors:

- **Transaction Length** - This key factor will dictate if the benchmark will be prone to benefit transactions running HTM or STM, depending on whether transactions exceed or not the capacity limitations of the underlying HTM implementation.

- **Update ratio** - Percentage of writes per transaction, this parameter will not only affect the contention of the workload but also will generate more transactional aborts, since transactions will abort due to conflicts with the writes of other transactions.

- **Switch of the partitions** - Since DMP-TM relies on the premise that several reference TM benchmarks present the memory layout that is easily divided in partitions [36], thus using that characteristic to run transactions using HTM and STM in independent partitions. It relies on the memory protection mechanism, namely mprotect() to ensure synchronization between transactions that step on partitions assigned to the other type of transactions. We use the parameter time_to_change_partitions that specifies the periodicity of the change of the type of the partitions. The smaller the time to switch the partitions, the greater the overhead incurred to DMP-TM, since it would be needed to apply more system calls to adapt to the new partitioning scheme.

Next, we used multiple workloads representative of real-life applications, such as the ones contained STAMP benchmark suite, in order to stress the advantages of using DMP-TM in real life applications.

In all the presented workloads we compare all the implementations using the following evaluation metrics:
• **Throughput** - Indicates the number of transactions committed per second.

• **Abort rate** - Indicates a ratio between the number of aborted and executed transactions, given by the formula `number_aborts/(number_aborts+number_commits)`. This metric is used to track and compare ratio of aborts between the TM implementations.

• **Breakdown of Commits** - Indicates the percentage of transactions that committed running HTM, STM or that have used the fallback mechanism.

• **Number of System Calls** - Since DMP-TM relies on `mprotect()` to synchronize the execution of transactions running HTM and transactions running STM, it is important to track the number of system calls issued, as this is an evidence of how precise the initial partitioning is.

All the 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 4.5.5 and the compiler used is GCC 6.2.1 with -O2 optimization level. The reported results represent the average of 5 runs.

For the cases that the number of threads exceeds the number of physical cores, we start to pin more than one thread to the same physical core, leading to have worse performance because different threads share the same processor, thus the same cache lines. If the memory is not aligned with the cache line size, two different variables could end up on the same cache line. This means that if two threads read/write two variables on the same cache line, then one of them is invalidated, further aborting transactions running hardware and the processor is forced to use concurrency mechanisms to access those cache lines. By aligning the variables to the cache line size, we avoid false sharing as two variables are guaranteed to reside on different cache lines.

Also, we tuned the value of maximum changes of protection of 5. This way, the maximum partition’s change that we allow is 5, after that, we use the fallback mechanism.

### 4.2 Experiments with Synthetic Benchmarks

In order to assess the effectiveness of DMP-TM in diverse, yet identifiable workload settings, we rely on a synthetic benchmark based on two pool of pages, HTM and STM pool, where each page represents a set of buckets containing various elements. Each page, depending on the pool that it belongs, is composed by `l` buckets with each bucket containing `e` elements. A page belonging to the HTM pool is composed by `l = 10 240` buckets with each bucket containing `e = 32` elements not ordered, being each element a `long`. This way it is expected to be benefited the execution of transactions running HTM in deterioration of transactions running STM, since if a transaction have to go through all the elements of the bucket the size of each bucket will not exceed the cache capacity which is 8KB (see table 2.1), i.e., 1024 elements, because each element is 8 bytes long. On opposite side, if the page belongs to STM pool, then the page is composed by `l = 176` buckets with each bucket containing `e = 1024` elements not ordered, this hashmap is expected to benefit transactions running STM in deterioration of transactions running HTM, since if a transaction reads all the elements of the bucket will likely to exceed the cache’s capacity and abort due to capacity aborts.
There are three main parameters that we use to tune the benchmark in order to exert precise control over the workload characteristics and gain deeper understanding over the behavior of DMP-TM over certain workload characteristics in comparison with the state of the art TM implementations. The parameters are the following:

- **p_bias** - indicates the percentage of operations issued on pages that belong to the STM pool, if \( p_{bias} \) equals to 0\%, then all operations will be issued to pages belonging to HTM pool and by opposition, if \( p_{bias} \) equals to 100\% then all the operations will be issued to pages belonging to the STM pool. This way is possible to control the transaction length, since the upper limit is 32 elements for transactions issuing operations on pages belonging to the HTM pool and 1024 elements for transactions issuing operations on pages belonging to the STM pool, being the former case beneficial for transactions running HTM, because they are faster than transactions running STM by not incurring extra instrumentation, and the latter for transactions running STM because the number of elements will likely induce capacity aborts by overflowing the cache’s capacity.

- **p_op** - indicates the percentage of the update operations issued, if \( p_{op} \) equals 0\% then all the operations issued are reads and by opposition, if \( p_{op} \) equals to 100\%, then all the operations issued are writes.

- **time to change partitions** - indicates the periodicity in which a page changes from one pool to the other, thus changing the composition of each bucket, i.e., if a page belongs to the HTM pool it is composed by \( l = 10 \text{ 240} \) buckets, with each bucket containing \( e = 32 \) elements and changes to being composed by \( l = 176 \) buckets with each bucket containing \( e = 1024 \) elements, thus changing the type of transactions that it benefits. The purpose of this parameter is to exercise how resilient is DMP-TM to partitions change. The usage of this parameter produce a workload particularly challenging for DMP-TM, as a change of a page from either one pool to the other incur one system call \( mprotect() \); in the case the page transits from STM pool to the HTM pool, transactions running HTM try to access the page and if they do not have the access right, they have to issue a system call in order to restore the protection the page; in a similar way, if the page transits from the HTM pool to the STM pool and the transaction checks that the access rights are not the ones required for it to issue the operation, either because it wants to issue a read and the protection for transaction HTM is Read/Write or because it wants to issue a write and the protection for the transactions running HTM is either Read or Read/Write, it has to issue a system call in order to revoke the access needed for the operation. The workloads generated by using this parameter are intended to stress how much system calls can DMP-TM handle.

The logic of the workload is the following: First are generated random numbers in order to decide which pool will be used and also which type of operation (read or write) will be issued, the generation of random number is done by using \( \text{random}_r() \), a version of \( \text{random}() \) that is thread_safe, i.e., generates independent, reproducible sequence of random numbers for different threads.

In this microbenchmark, we initially assign 40 pages to the HTM pool and 22 pages to the STM pool in a random way. This values were experimentally tested in order to grant scalability for both HTM and STM, in each respective pool.

Then, according to the random numbers obtained and the \( p_{bias} \) and \( p_{op} \) defined at the beginning, the thread will pick a random page from either the HTM or STM pool and issue a read or write operation.
in it. If it is a read operation, it is chosen a random number between in the range of [0-Pool Population], being Pool Population the possible number of values present in the bucket. We use Pool Population 100 times the number of elements in a bucket. Then, it will calculate the bucket in which the element would be present, by simply using the formula 

\[ \text{bucket to search} = \frac{\text{value to find}}{\text{total number of buckets in page}} \]

and go through all the elements in the bucket, if it finds the value that it is looking for, it stops, if not it continues until the end of the bucket.

If it is a write operation, it will generate two random numbers between the range of [0-Pool Population], then it will find the bucket which is supposed to be the first value by using the formula stated before and will read through all the elements, if it finds the element, it writes the second value instead of the first, if not, it continues until the end of the bucket. By reaching the end of the bucket, it writes the second value in a random position.

The usage of \textit{time to change partitions} is slightly more complex, the benchmark receives as initial parameter the time that it should run and the \textit{time to change partitions}, this value is used as the input for the random number generator. Instead of using always the same fixed value for changing the pool of a certain page, it is generated a random number using an exponential distribution with the input:

\[
\text{input} = \frac{1}{\text{time to change partitions}}
\]

to simulate a real application where is not expected that pages move all at the same time from one pool to the other. It is maintained by the system a structure that indicates the last time that the page has moved from one heap to the other and also the expected \textit{time to change the partition}. We used a special thread, named \textit{worker thread}, which job is to go through all the times elapsed of each page and check if they already surpassed the time limit, if they do, the \textit{worker thread} signals to the transactions that no new transaction can issue operations on that page and waits for the transactions that are currently issuing operations on the page to finish their execution. Then, it generates a new random \textit{time to change the partition} using the exponential distribution and changes the pool of which the page belongs.

Therefore, we consider 5 different workload scenarios, differing by the likelihood accessing pages in the HTM pool, namely 0%, 1%, 2%, 5%, 50% and 100%. We present the scenarios that use \(p_{bias}\) both equal to 0% and 100% to stress that DMP-TM, in the former case has throughput comparable to HTM-SGL and in the latter one to show the overhead incurred by the instrumentation on top of TinySTM.

For the experiments with \(p_{bias}\) equals to 1%, 2%, 5% and 50%, we want to stress that whenever the workload has transactions accessing pages from both pools and the pages do not move between pools, DMP-TM, by recurring to compiler static analysis, has knowledge about in which pool is the page being accessed contained, thus using the correct type of transactions in the operation. Because the partitions are heterogeneous and do not move and by using the correct type of transactions recurring to compiler analysis, DMP-TM achieves the best performance of the TM implementations using in the experiment.

For each value of \(p_{bias}\), we present three different workloads where we variate the percentage of writes, i.e., \(p_{op}\), respectively 10%, 50% and 90%.

In subsection 4.2.1, we first tested all of the variants of the algorithm described in 3.2.8 in order to assess which are the ones with the best throughput with different workloads: we first used \(p_{bias} = 1\%\), \(p_{op} = 10\%\) and then \(p_{bias} = 2\%\), \(p_{op} = 90\%\). Then, in subsection 4.2.2, we compared the best two
variants with the state of the art TM implementations in workloads generated by $p_{\text{bias}}$ equals to 0\%, 1\%, 2\%, 5\%, 50\% and 100\% and $p_{\text{op}}$ equals to 10\%, 50\% and 90\%.

### 4.2.1 Comparison of different variants of the algorithm

We first compared all of the developed variants in order to state which of the variants of the algorithm had higher throughput to later compare the best variants against other state of the art TM implementations.

The results of the experiments done with all the variants described in subsection 3.2.8, namely the variant that fallback to acquiring first all the per-thread locks and then the single global lock using transactions running HTM, the variant that fallback to acquiring first the single global lock and then all the per-thread locks using transactions running HTM, the variant that fallback to acquire first the single global lock and then all the per-thread locks using the compare-and-swap directive, the variant that fallback to using TinySTM without any extra-overhead, the variant that first acquires all the per-thread locks and then the single global lock using the directive compare-and-swap and finally the variant that fallback to using non-instrumented execution regulated by reading a single variable using memory barriers are presented in figure 4.1, we tested the execution of the benchmark using 1, 2, 4, 8, 16, 32, 64 and 80 threads and measured the throughput in the form of number of transactions per second.

Figure 4.1 (a) and figure 4.1 (c) present the experiment done with the probability of accessing pages in the STM pool, $p_{\text{bias}}$, equals to 1\% and the percentage of writes equals to 10\%, which produces a workload highly probable to benefit transactions running HTM because 99\% of the times are accessed pages from the HTM pool, thus having fewer elements per bucket.

By analyzing the results, it is possible to check that the trend is to scale up to 32 threads and then the throughput tends to hinder for executions with 64 and 80 threads. In particular, the variant of the algorithm that achieves higher throughput up to 32 thread is the one that first acquires the single global lock and then all the per-thread locks in order to fallback to an uninstrumented execution. Because transactions running HTM only check if the single global lock is free and transactions running STM signal their execution to a per-thread lock, thus the only case where the two types of transaction share the same locks is when the fallback is used, the abort rate is very low. This variant achieves an average speedup of 1.16x in comparison with other variants up to 32 threads, however after 32 threads the majority of the solutions tend to decrease its throughput because not only the contention increases with a larger number of threads but also in practice all the processors share more than one thread, thus a cache, which causes aborts. This increase of aborts have a high impact on the throughput, since for all the variants, with exception of the solution that uses TinySTM as the fallback, all the threads must stop in order to run the fallback mechanism.

However, this is not the case of the solution that fallback to running uninstrumented TinySTM, after 32 threads the increase of aborts of transactions running HTM leads to acquiring the fallback mechanism, that aborts the running HTM transactions and disallows the activation of new transactions running HTM, however this fallback mechanism is compatible with transactions running STM, which is the reason of not having throughput losses from 32 to 80 threads. Surprisingly, the solution that fallback to acquiring first the single global lock and then all the per-thread locks using HTM transactions has
(a) Throughput of experiment on hashmap with $p_{bias}=1\%$ and $p_{op}=10\%$

(b) Throughput of experiment on hashmap with $p_{bias}=2\%$ and $p_{op}=90\%$

(c) Abort rate of experiment on hashmap with $p_{bias}=1\%$ and $p_{op}=10\%$

(d) Abort rate of experiment on hashmap with $p_{bias}=2\%$ and $p_{op}=90\%$

Figure 4.1: Comparison of different variants of the algorithm using the synthetic benchmark with time to change partitions = infinity

The lower speedups in this experiment, since after 16 threads the higher number of aborts leads to the abort of every transaction.

Figure 4.1 (b) and figure 4.1 (d) present an experiment with the probability of accessing pages in the STM pool, $p_{bias}$, equals to 2% and the percentage of writes equals to 90%, which produces a workload with higher probability of benefiting transactions running STM than the workload described before and because the percentage of writes is much higher, more prone to have higher abort rate, which in practice will exercise the ability of the solutions to use the fallback and to deal with high contention scenarios.

By analyzing the figure is possible to infer that in the generated workload, the trend is to every variant of the algorithm scale up to 16 threads, with the exception of both the variant that acquires all the per-thread locks first, killing all the running transactions executing STM and the variants that acquire first the SGL using transactions running HTM. It should be noted that however all the other variants scale up to 16 threads, the variants that performed best in the latter experiment achieves speedups in the order of 1.4x in this experiment compared with the two other experiments that also scale: the solution that acquires first all the per-thread locks using transactions and the solution that uses memory barriers to
write a variable indicating the state. After 16 threads, the solution with higher throughput, i.e. the one that fallback to acquiring first the SGL and then all the per-thread locks starts to have higher number of aborts, a trend shared by all the other variants but at 32 threads. In a similar way to the latest experiment, after 32 threads the solution that fallback to the execution of non-instrumented TinySTM begins to have higher abort rate and because of that, fallback to a non-instrumented version of TinySTM and only have a decrease of 10% in the throughput in comparison to the average of 40% experienced by the other variants.

In conclusion, the analysis carried in this subsection 4.2.1 highlighted that there are two different variants of the algorithm that might achieve good results in different scenarios, we chose this two variants to be compared with the state of the art TM implementations:

- **Solution that acquires first the SGL and then all the per-thread locks using atomic operations** - this solution achieves the best throughput from all the variants proposed for workloads encompassing low usage of pages in the STM pool up to 32 threads. In the rest of the document this solution will be referred as DMP-SGL.

- **Solution that fallback to uninstrumented TinySTM** - this solution achieved consistently better throughput after 32 threads in all workloads and achieves the same throughput of DMP-SGL up to 32 threads for workloads characterized by high percentage of updates, namely 90%. In the rest of the document this solution will be referred as DMP-STM.

### 4.2.2 Partitionable Hashmap with no Partition Change

In this subsection, we compare both DMP-SGL and DMP-STM with the state of the art TM implementations, we want to stress the advantages of using DMP against HTM, STM [13; 21] and Hybrid TM [12] implementations.

We manipulated $p_{bias}$ in order to generate 6 different workloads, $p_{bias}$ equals to 0%,1%,2%,5%, 50% and 100% and for each value of $p_{bias}$ we present the scenario with 3 different percentage of update transactions: $p_{op}=10\%, 50\%$ and 90%.

Figure 4.2 reports the execution of the workload with $p_{bias} = 0\%$, a scenario where only are accessed pages from the HTM pool, which generates a workload with high likelihood of benefiting transactions running HTM, while varying the number of threads. We used different values of $p_{op}$, namely 10%, 50% and 90% to induce higher percentage of writes in the generated workload. To achieve a scenario where pages do not move from one pool to another, we manipulated time to change partitions to be much higher than the time that took to run the microbenchmark.

The plots show that for this workload tuned with $p_{op} = 10\%$, which is a read dominated workload, HTM-SGL has slightly better throughput than DMP-SGL up to 32 threads and slightly better throughput than Hybrid-NOrec and DMP-STM up to 16 threads. After 16 threads both Hybrid-NOrec and DMP-STM start having aborts that hinder their performance, however because DMP-SGL has to acquire first the single global lock and then all the per-thread locks to execute the fallback mechanism, this serializes the execution of the workload, since it is taken more time to activate the fallback mechanism and to release the locks, which causes DMP-SGL to achieve speedups in the order of 3x against the other
TM implementations after 32 threads. Because the number of elements per bucket is so small, the overheads incurred by the instrumentation of each operation by the transactions running STM hinders the throughput, making the throughput on average 5 times lower than the other TM implementations. However, TinySTM in this specific workload, begins to outperform NOrec after 8 threads, achieving on average 2x speedups than the other state of the art STM implementation.

This specific workload with \( p_{op} \) equals to 50% and 90% generates results in where HTM-SGL scales up to 32 threads and have slightly better performance than the other TM implementations that use transactions running HTM, namely DMP-SGL, DMP-STM and Hybrid-NORec. Hybrid-NOrec after 16 threads has a crash in the throughput as it begins to have non-negligible abort rate that escalates up to 0.9, a similar trend shared by DMP-STM and HTM-SGL but for 32 threads. DMP-SGL, in a similar way to the workload generated by \( p_{bias} = 0\% \) and \( p_{op} = 10\% \), because it has to acquire more locks and also release, it serializes the execution and achieves better throughput. The STM implementations continue to achieve bad results, having an average speed loss of 4x comparing to the other solutions of the experiment.

Figure 4.3 captures the results obtained by adjusting the parameter \( p_{bias} \) equals to 2\%. In all the workloads obtained by varying the probability of writes, HTM-SGL scales up to 32 threads, being
Throughput of experiment on hashmap with $p_{bias}=2\%$ and $p_{op}=10\%$

Throughput of experiment on hashmap with $p_{bias}=2\%$ and $p_{op}=50\%$

Throughput of experiment on hashmap with $p_{bias}=2\%$ and $p_{op}=90\%$

Abort rate of experiment on hashmap with $p_{bias}=2\%$ and $p_{op}=10\%$

Abort rate of experiment on hashmap with $p_{bias}=2\%$ and $p_{op}=50\%$

Abort rate of experiment on hashmap with $p_{bias}=2\%$ and $p_{op}=90\%$

Figure 4.3: Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using probability of accessing the STM pool = 2\%.

Figure 4.4 presents the results of the experiment done with $p_{bias}=5\%$ and $p_{op}$ equals to $10\%$, $50\%$ and $90\%$. It is to be noted that as the value of $p_{bias}$ increases, the overall throughput is lower and...
transactions running STM begin to outperform transactions running HTM, because of the capacity aborts incurred by the reading of higher number of elements. HTM-SGL still achieves the higher throughput up to 8 threads and then it begins to have consistently lower throughput when the number of threads increase. Hybrid-NOrec and NOrec are not competitive in this test, since Hybrid-NOrec has abort rate in the order of 0.4 after 16 threads with \( p_{op} \) equals to 10%, average abort rate of 0.55 whenever \( p_{op} \) equals to 50% after 8 threads and average abort rate of 0.6 whenever \( p_{op} \) equals to 90% after 8 threads. DMP-STM and DMP-SGL achieves the best throughput in this test, being DMP-SGL the best variant up to 32 threads and then continuously have performance penalties up to 80 threads, where it is outperformed by TinySTM. DMP-STM maintains its throughput after 32 threads becoming the best variant with exception of the case that \( p_{op} \) equals to 10%, where it has 0.16x less speedups than TinySTM because TinySTM achieves better performance with low contention scenarios. TinySTM with the increase of \( p_{bias} \), begins to scale better and after 16 threads, in the case of \( p_{op} \) equals to 10% and 50%, and after 32 threads in the case of \( p_{op} \) equals to 90%, outperforms HTM-SGL.

Figure 4.5 presents the results achieved by using \( p_{bias} \) equals to 50%, with \( p_{op} \) equals to 10%, 50% and 90%. By analyzing this figure is possible to infer that HTM-SGL only is the better choice up to 4 threads, achieving speedups in the order of 2x against all other implementations. After 4 threads the best throughput is consistently achieved by TinySTM because 50% of the operations are issued to the STM pool and this operation have 32 times more duration than operations issued on pages of the HTM pool. Because of that the workload benefits transactions running STM, more precisely TinySTM and NOrec, being DMP-SGL the second best choice after 16 threads, where NOrec has a performance penalty. The workload generated by \( p_{op} \) equals to 90% produce a high contention workload, which is unlikely to benefit NOrec, in fact in this workload, both NOrec and Hybrid-NOrec achieve the worse throughput values of the test, being Hybrid-NOrec slightly worse than NOrec. DMP-SGL takes advantage of this and becomes the second best implementation, having only on average 25% loss from TinySTM compared with 50% with lower values of \( p_{op} \).

Figure 4.6 presents the results of the experiment with \( p_{bias} \) equals to 100%, this workload only generates operations in pages belonging to STM pool. This workload follows a similar trend then the workload with \( p_{bias} \) equals to 50%, with the difference that for \( p_{op} \) equals to 10% NOrec achieves the best throughput for 8 and 16 threads because it has higher throughput in workloads characterized by lower percentage of updates. After that, TinySTM achieves better throughput than all other TM implementations in the experiments. For \( p_{op} \) equal to 50% and 90%, NOrec has significant performance losses, which makes TinySTM the TM implementation with the best throughput after 8 threads. DMP-SGL and DMP-STM are, respectively second and third best after 8 threads with in average, respectively 0.76x and 0.46x of the throughput of TinySTM.
(a) Throughput of experiment on hashmap with $p_{\text{bias}}=5\%$ and $p_{\text{-op}}=10\%$

(b) Throughput of experiment on hashmap with $p_{\text{bias}}=5\%$ and $p_{\text{-op}}=50\%$

(c) Throughput of experiment on hashmap with $p_{\text{bias}}=5\%$ and $p_{\text{-op}}=90\%$

(d) abort rate of experiment on hashmap with $p_{\text{bias}}=5\%$ and $p_{\text{-op}}=10\%$

(e) abort rate of experiment on hashmap with $p_{\text{bias}}=5\%$ and $p_{\text{-op}}=50\%$

Figure 4.4: Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using probability of accessing the STM pool = 5%

4.2.3 Partitionable Hashmap with Partition Change

This subsection presents the study carried out by using values of `time_to_change_partitions` different than infinity. We tuned `time_to_change_partitions` so that in a run of 5 seconds, the pages changes from one pool to the other following an exponential distribution with average value of 2 seconds. By using an exponential distribution, we intended to mimic a real-life application, where the pages change at different times.

By analysing Figure 4.7 and tables 4.2, 4.3 and 4.4 is possible to infer that whenever the partitions are not disjoint, the usage of system calls to guarantee progress incur a large overhead. However, as possible to see in figure 4.7 (b) and (c), the overhead of issuing on average 46 system calls by DMP-SGL and 49 by DMP-STM are not large enough to hamper the performance. TinySTM shows to adapt well to partition changes, and on average, it has 1.5x speedups to the second contender, DMP-SGL. DMP-SGL can still have higher throughput than the rest of the implementations, except TinySTM, because the percentage of transactions that use the fallback mechanism is at most 1%, which makes the solution delivering optimal performance independently of the pool accessed. It is to be noted that, in workloads with more than 50% of writes, DMP-TM shows having up to 6x throughput than the Hybrid TM contender, Hybrid-NOrec [12]. DMP-STM on average has throughput losses in the order of 0.68x
(a) Throughput of experiment on hashmap with $p_{bias}=50\%$ and $p_{op}=10\%$

(b) Throughput of experiment on hashmap with $p_{bias}=50\%$ and $p_{op}=50\%$

(c) Throughput of experiment on hashmap with $p_{bias}=50\%$ and $p_{op}=90\%$

(d) abort rate of experiment on hashmap with $p_{bias}=50\%$ and $p_{op}=10\%$

(e) abort rate of experiment on hashmap with $p_{bias}=50\%$ and $p_{op}=50\%$

(f) abort rate of experiment on hashmap with $p_{bias}=50\%$ and $p_{op}=90\%$

Figure 4.5: Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using probability of accessing the STM pool = 50%

compared to DMP-SGL, but however, still has average speedups of 1.15x compared with Hybrid-NOrec. Also, the analysis of figure 4.8 highlighted that with the lowering of the time to change partitions comes an inherent overhead that lowers the throughput.

<table>
<thead>
<tr>
<th>STAMP Benchmark</th>
<th>Hybrid-NOrec</th>
<th>DMP-STM</th>
<th>DMP-SGL</th>
<th>TinySTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>HTM (%)</td>
<td>STM (%)</td>
<td>Fallback (%)</td>
<td>HTM (%)</td>
</tr>
<tr>
<td>2</td>
<td>67</td>
<td>0</td>
<td>33</td>
<td>66</td>
</tr>
<tr>
<td>4</td>
<td>68</td>
<td>0</td>
<td>32</td>
<td>68</td>
</tr>
<tr>
<td>8</td>
<td>68</td>
<td>0</td>
<td>32</td>
<td>66</td>
</tr>
<tr>
<td>16</td>
<td>66</td>
<td>0</td>
<td>34</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>65</td>
<td>0</td>
<td>35</td>
<td>62</td>
</tr>
<tr>
<td>64</td>
<td>66</td>
<td>0</td>
<td>34</td>
<td>60</td>
</tr>
<tr>
<td>80</td>
<td>67</td>
<td>0</td>
<td>33</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 4.2: Breakdown of commit types of state of art TM implementations with the probability of accessing a page in the STM pool=2% with $p_{op} = 10\%$ with pages changing pool every 2 seconds

### 4.3 STAMP benchmark

In this subsection we present three different experiments conducted with STAMP benchmark. This experiments are intended to stress the benefits of using DMP-STM and DMP-SGL in real-world appli-
Throughput of experiment on hashmap with $p_{bias}=100\%$ and $p_{op}=10\%$

Throughput of experiment on hashmap with $p_{bias}=100\%$ and $p_{op}=100\%$

Throughput of experiment on hashmap with $p_{bias}=100\%$ and $p_{op}=90\%$

Abort rate of experiment on hashmap with $p_{bias}=100\%$ and $p_{op}=10\%$

Abort rate of experiment on hashmap with $p_{bias}=100\%$ and $p_{op}=50\%$

Abort rate of experiment on hashmap with $p_{bias}=100\%$ and $p_{op}=90\%$

Figure 4.6: Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using probability of accessing the STM pool = 100%

...
different workloads: vacation low, a workload with low contention, and vacation high, a workload with high contention. In addition to vacation low and vacation high workloads we generated a third one using the following parameters: \(-n5 -q60 -u90 -r16384 -t4096\). The generated workload has higher number of queries per client, which tests the workloads with more contention than vacation high.

By analyzing figure 4.9 (a), (d) and (g) is possible to infer that in vacation low, HTM-SGL scales up to 4 threads, closely followed by NOrec, that has higher throughput due to the low contention present in this workload, with average throughput difference of 5%. Then, after 4 threads, both HTM-SGL and NOrec have performance penalties due to the increase of the abort rate, which goes from roughly 0.017 with 8 threads to almost 0.025 with 8 threads, and continue to decrease their throughput up to 80 threads. At 16 threads, DMP-SGL, TinySTM and DMP-STM begin to outperform all the other TM implementations and remain that way for the rest of the thread counts; DMP-SGL achieves the best throughput of the test after 16 threads, having average 1.13 speedups in comparison with TinySTM, we achieve this by running software transactions whenever are done reservations or whenever the administrator add a new item to the database, all other operations, namely the delete of a customer and the calculation of a bill of a certain client are calculated by using transactions running HTM. DMP-STM achieves the third best throughput in the study, having 0.12x throughput penalty compared with the second best, TinySTM. By checking figure 4.9 (g) and table 4.5 is possible to infer that with this scheme, are issued roughly 160 system calls to prevent conflicts and that on average for both DMP-STM and DMP-SGL are issued on
average 99% of transactions running STM with the rest of the transactions being divided by transactions running HTM and transactions using the fallback mechanism.

By analyzing figure 4.9 (b), (e) and (h) is possible to infer that in vacation-high, the workload is prone to benefit HTM-SGL and NOrec up to 4 threads, which after that thread count begin to have some performance penalties. DMP-SGL has the worst throughput in this experiment, since the average commit of 30% of transactions executing the fallback mechanism hinder the throughput. DMP-STM achieves the second best throughput in the experiment, because of the higher abort rate, transactions revert to the execution of TinySTM, which greatly improves the performance of this experiment. TinySTM achieves the best throughput in this experiment with 1.3x speedups in comparison with the second best, DMP-STM. Hybrid NOrec achieves a medium performance in this experiment, being better than HTM-SGL and DMP-SGL but being worse than TinySTM, DMP-STM and NOrec. DMP-STM and DMP-SGL issue an average of 200 system calls during their execution to synchronize the execution of transactions running STM and transactions running HTM.

By analyzing figure 4.9 (c), (f) and (i) is possible to infer that in the proposed scenario of vacation with more queries per-client, NOrec achieves the best throughput up to 8 threads and after it, the workload is dominated by the proposed solutions that, in the case of DMP-SGL, achieve on average 1.3x speedups in comparison with TinySTM and Hybrid-NOrec and on average 3.25x speedups in comparison with HTM-SGL.

### 4.3.2 Genome

Genome represents the process of taking a large number of DNA segments and matching them to reconstruct the original source genome. This benchmark has two phases: the first phase uses an hash-set on the pool of duplicate DNA segments in order to create a set of unique segments. In the second phase, it tries to take a segment from the pool and try to add it to a set of currently matched segments. Transactions are used in both phases of the algorithm to guarantee that the changes to the
Table 4.5: Breakdown of commit types of state of art TM implementations running vacation low

<table>
<thead>
<tr>
<th>Threads</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>79</td>
<td>0</td>
<td>30</td>
<td>54</td>
<td>0</td>
<td>46</td>
<td>0.7</td>
<td>99</td>
<td>0.3</td>
<td>0.8</td>
<td>99</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>67</td>
<td>0</td>
<td>33</td>
<td>53</td>
<td>0</td>
<td>47</td>
<td>0.8</td>
<td>99</td>
<td>0.2</td>
<td>0.8</td>
<td>99</td>
<td>0.2</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>0</td>
<td>36</td>
<td>53</td>
<td>0</td>
<td>47</td>
<td>0.9</td>
<td>99</td>
<td>0.1</td>
<td>1.0</td>
<td>98</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>54</td>
<td>0</td>
<td>40</td>
<td>54</td>
<td>0</td>
<td>46</td>
<td>0.7</td>
<td>99</td>
<td>0.3</td>
<td>1.1</td>
<td>98</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>49</td>
<td>0</td>
<td>51</td>
<td>38</td>
<td>0</td>
<td>62</td>
<td>0.5</td>
<td>99</td>
<td>0.5</td>
<td>0.9</td>
<td>98</td>
<td>1.1</td>
</tr>
<tr>
<td>32</td>
<td>52</td>
<td>0</td>
<td>48</td>
<td>38</td>
<td>0</td>
<td>62</td>
<td>0.5</td>
<td>99</td>
<td>0.5</td>
<td>0.9</td>
<td>98</td>
<td>1.1</td>
</tr>
<tr>
<td>64</td>
<td>60</td>
<td>0</td>
<td>40</td>
<td>45</td>
<td>0</td>
<td>55</td>
<td>0.6</td>
<td>99</td>
<td>0.4</td>
<td>0.9</td>
<td>98</td>
<td>1.1</td>
</tr>
<tr>
<td>80</td>
<td>61</td>
<td>0</td>
<td>39</td>
<td>45</td>
<td>0</td>
<td>54</td>
<td>0.7</td>
<td>99</td>
<td>0.3</td>
<td>0.9</td>
<td>98</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Table 4.6: Breakdown of commit types of state of art TM implementations running vacation High

<table>
<thead>
<tr>
<th>Threads</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>41</td>
<td>0</td>
<td>59</td>
<td>33</td>
<td>0</td>
<td>67</td>
<td>28</td>
<td>10</td>
<td>62</td>
<td>31</td>
<td>10</td>
<td>59</td>
</tr>
<tr>
<td>2</td>
<td>39</td>
<td>0</td>
<td>61</td>
<td>32</td>
<td>0</td>
<td>68</td>
<td>26</td>
<td>10</td>
<td>62</td>
<td>31</td>
<td>10</td>
<td>59</td>
</tr>
<tr>
<td>4</td>
<td>38</td>
<td>0</td>
<td>62</td>
<td>32</td>
<td>0</td>
<td>68</td>
<td>29</td>
<td>10</td>
<td>61</td>
<td>30</td>
<td>10</td>
<td>59</td>
</tr>
<tr>
<td>8</td>
<td>37</td>
<td>0</td>
<td>63</td>
<td>31</td>
<td>0</td>
<td>69</td>
<td>27</td>
<td>10</td>
<td>63</td>
<td>30</td>
<td>10</td>
<td>63</td>
</tr>
<tr>
<td>16</td>
<td>33</td>
<td>0</td>
<td>67</td>
<td>24</td>
<td>0</td>
<td>76</td>
<td>30</td>
<td>9</td>
<td>61</td>
<td>27</td>
<td>9</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>35</td>
<td>0</td>
<td>65</td>
<td>19</td>
<td>0</td>
<td>81</td>
<td>35</td>
<td>10</td>
<td>55</td>
<td>20</td>
<td>10</td>
<td>70</td>
</tr>
<tr>
<td>64</td>
<td>41</td>
<td>0</td>
<td>59</td>
<td>20</td>
<td>0</td>
<td>80</td>
<td>36</td>
<td>10</td>
<td>54</td>
<td>20</td>
<td>10</td>
<td>70</td>
</tr>
<tr>
<td>80</td>
<td>44</td>
<td>0</td>
<td>57</td>
<td>22</td>
<td>0</td>
<td>76</td>
<td>36</td>
<td>10</td>
<td>54</td>
<td>21</td>
<td>10</td>
<td>69</td>
</tr>
</tbody>
</table>

By checking figure 4.10 is possible to check that HTM-SGL has higher throughput up to 2 threads and after that DMP-STM achieves the best throughput in all the study, achieving speedup gains in the order of 1.2x until 64 threads, where it begins to have similar throughput of TinySTM. DMP-SGL is not competitive, achieving the third best throughput due to the fact that it has roughly 0.3 abort rate and 1% of transactions using the fallback mechanism, which hinders the performance. Hybrid-NOrec has the worst throughput in this experiment, mainly because of its high abort rate, which causes falling back to NOrec and consequently the abort of other transactions running HTM. DMP-SGL commits using 97% of transactions running HTM, 2% of transactions running STM and 1% of transactions using the fallback mechanism. DMP-STM commits using 95% of transactions running HTM, 4% of transactions running STM and 1% of transactions using the fallback mechanism; we achieve this by running transactions executing STM in the first phase of the algorithm and running transactions executing HTM in the second phase of the algorithm.

4.3.3 Intruder

Intruder is a signature-based network intrusion detection systems (NIDS) that scan network packets for matches against a known set of intrusion signatures. Network packets are processed in a parallel way and have to go through three main phases: capture, reassembly and detection. The capture and reassembly phases are enclosed in transactions to guarantee that concurrent modifications to the packets are safe.

By analyzing figure 4.11 and table 4.9 is possible to infer that the workload generated is highly prone to benefit transactions running HTM. HTM-SGL and DMP-SGL achieve the best throughput up to 16 threads, after that NOrec outperforms them. DMP-STM achieves the worst throughput of the testbed, mainly because of its high abort rate. We did not found a good partitioning in the memory layout in this benchmark, this way this experiment is intended to show that for workloads prone to benefit HTM-SGL,
<table>
<thead>
<tr>
<th>Threads</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>31</td>
<td>0</td>
<td>69</td>
<td>24</td>
<td>0</td>
<td>76</td>
<td>0.9</td>
<td>99</td>
<td>0.1</td>
<td>0.9</td>
<td>99</td>
<td>0.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>0</td>
<td>70</td>
<td>25</td>
<td>0</td>
<td>75</td>
<td>0.9</td>
<td>99</td>
<td>0.1</td>
<td>1</td>
<td>99</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>0</td>
<td>70</td>
<td>26</td>
<td>0</td>
<td>74</td>
<td>0.9</td>
<td>99</td>
<td>0.1</td>
<td>1</td>
<td>99</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>30</td>
<td>0</td>
<td>70</td>
<td>26</td>
<td>0</td>
<td>74</td>
<td>0.9</td>
<td>99</td>
<td>0.1</td>
<td>1</td>
<td>99</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>28</td>
<td>0</td>
<td>72</td>
<td>20</td>
<td>0</td>
<td>80</td>
<td>0.8</td>
<td>99</td>
<td>0.2</td>
<td>1</td>
<td>99</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>30</td>
<td>0</td>
<td>70</td>
<td>15</td>
<td>0</td>
<td>85</td>
<td>0.7</td>
<td>99</td>
<td>0.3</td>
<td>1</td>
<td>99</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>35</td>
<td>0</td>
<td>66</td>
<td>16</td>
<td>0</td>
<td>84</td>
<td>0.5</td>
<td>99</td>
<td>0.5</td>
<td>1</td>
<td>99</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>36</td>
<td>0</td>
<td>66</td>
<td>17</td>
<td>0</td>
<td>83</td>
<td>0.3</td>
<td>99</td>
<td>0.2</td>
<td>1</td>
<td>99</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7: Breakdown of commit types of state of art TM implementations running **vacation with different number of queries per client**

DMP can have performance comparable to it.
Figure 4.8: Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using hashmap with partition change of 1 and 0.5 seconds

<table>
<thead>
<tr>
<th>Threads</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
<th>DMP-SGL</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
<th>DMP-STM</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Fallback (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>91</td>
<td>0</td>
<td>9</td>
<td>31</td>
<td>0</td>
<td>69</td>
<td>97</td>
<td>2</td>
<td>1</td>
<td>95</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>91</td>
<td>0</td>
<td>9</td>
<td>31</td>
<td>0</td>
<td>69</td>
<td>97</td>
<td>2</td>
<td>1</td>
<td>95</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>92</td>
<td>0</td>
<td>8</td>
<td>32</td>
<td>0</td>
<td>68</td>
<td>97</td>
<td>2</td>
<td>1</td>
<td>95</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>92</td>
<td>0</td>
<td>8</td>
<td>32</td>
<td>0</td>
<td>68</td>
<td>97</td>
<td>2</td>
<td>1</td>
<td>95</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>90</td>
<td>0</td>
<td>10</td>
<td>30</td>
<td>0</td>
<td>70</td>
<td>97</td>
<td>2</td>
<td>1</td>
<td>95</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>89</td>
<td>0</td>
<td>11</td>
<td>30</td>
<td>0</td>
<td>70</td>
<td>97</td>
<td>2</td>
<td>1</td>
<td>95</td>
<td>4</td>
</tr>
<tr>
<td>64</td>
<td>89</td>
<td>0</td>
<td>11</td>
<td>30</td>
<td>0</td>
<td>71</td>
<td>97</td>
<td>2</td>
<td>1</td>
<td>95</td>
<td>4</td>
</tr>
<tr>
<td>80</td>
<td>90</td>
<td>0</td>
<td>10</td>
<td>29</td>
<td>0</td>
<td>71</td>
<td>97</td>
<td>2</td>
<td>1</td>
<td>95</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.8: Breakdown of commit types of state of art TM implementations running Genome
(a) Throughput of experiment using vaca-

(b) Throughput of experiment using vaca-

(c) Throughput of experiment using vaca-

(d) abort rate of experiment using vaca-

(e) abort rate of experiment using vaca-

(f) abort rate of experiment using textit-

(g) Number of system calls issued in ex-

(h) Number of system calls issued in ex-

(i) Number of system calls issued in ex-

Figure 4.9: Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using vacation benchmark

Table 4.9: Breakdown of commit types of state of art TM implementations running Intruder

<table>
<thead>
<tr>
<th>Threads</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Failback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Failback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Failback (%)</th>
<th>HTM (%)</th>
<th>STM (%)</th>
<th>Failback (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>70</td>
<td>0</td>
<td>30</td>
<td>70</td>
<td>0</td>
<td>30</td>
<td>58</td>
<td>0</td>
<td>42</td>
<td>70</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>69</td>
<td>0</td>
<td>31</td>
<td>69</td>
<td>0</td>
<td>31</td>
<td>57</td>
<td>0</td>
<td>43</td>
<td>70</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>69</td>
<td>0</td>
<td>31</td>
<td>67</td>
<td>0</td>
<td>33</td>
<td>56</td>
<td>0</td>
<td>44</td>
<td>69</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>8</td>
<td>69</td>
<td>0</td>
<td>31</td>
<td>60</td>
<td>0</td>
<td>40</td>
<td>57</td>
<td>0</td>
<td>43</td>
<td>69</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>16</td>
<td>69</td>
<td>0</td>
<td>31</td>
<td>60</td>
<td>0</td>
<td>40</td>
<td>57</td>
<td>0</td>
<td>43</td>
<td>68</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>32</td>
<td>69</td>
<td>0</td>
<td>31</td>
<td>60</td>
<td>0</td>
<td>40</td>
<td>57</td>
<td>0</td>
<td>43</td>
<td>68</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>69</td>
<td>0</td>
<td>31</td>
<td>60</td>
<td>0</td>
<td>40</td>
<td>57</td>
<td>0</td>
<td>43</td>
<td>68</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>96</td>
<td>69</td>
<td>0</td>
<td>31</td>
<td>60</td>
<td>0</td>
<td>40</td>
<td>57</td>
<td>0</td>
<td>43</td>
<td>68</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 4.9: Breakdown of commit types of state of art TM implementations running Intruder
Figure 4.10: Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using Genome benchmark
Figure 4.11: Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using intruder benchmark
5.1 Conclusions

Transactional Memory is an interesting paradigm since it allows the development of highly concurrent applications without having to use locks to synchronize the manipulation of shared data. Throughout the last two decades, TM has been target of intense research, since there are numerous workloads to which this paradigm could be applied.

The first implementations of this paradigm were in the form of a software library, called STM, which has progress and correctness guarantees but incurs an extra overhead on the execution of the code in order to keep track of the operations performed.

Hardware Transactional Memory, was recently introduced in the recent processors, namely the mainstream Intel Haswell and IBM Power8. This means that transactions executing in hardware may never commit (even in absence of concurrency) due to inherent limitations of hardware, such as transactions exceeding capacity of hardware or running prohibited instructions. This raises the need for a software fallback path to ensure forward progress [19].

In an effort to achieve the best of both implementations, researchers created Hybrid Transactional Memory [9; 12; 14; 30; 37], which concept is to run the faster HTM transactions whenever possible and fallback to the more costly STM whenever HTM cannot guarantee progress. A large number of papers have been published in the last years and although a promising concept, Hybrid TM incurs extra-overheads due to the maintenance of shared metadata in order to guarantee the correctness of the concurrent manipulation of both HTM and STM.

In the light of the analysis of the state of the art performed in the chapter 2, and relying on a recent study [36] that indicates the partitionability of the memory layout of several reference TM benchmarks, this thesis proposes a novel Hybrid TM implementation called Dynamic Memory Partitioning (DMP-TM). DMP-TM relies on the hardware/operating system's memory protection mechanisms, i.e., the mprotect() system call, to synchronize the shared accesses by HTM and STM.

DMP-TM divides the memory layout in pages and those pages are either accessed by HTM with no instrumentation, or using STM, which incurs an extra-overhead by changing the protection via system call for transactions running by HTM , with no instrumentation, or by STM, with minimal additional instrumentation, at least in absence of memory partitions’ violations. That way whenever one implementation accesses data in a page to which its access is not allowed, it receives an SIGSEGV that indicates that the page could be accessed by the other implementation and possibly causing a conflict and wrong behavior.
We designed an algorithm intended to minimize the costs of having to issue system calls and evaluated several fallback mechanisms. In the light of this study, this thesis proposes two variants of the algorithm: DMP-SGL and DMP-STM. DMP-SGL intends to use non-instrumented execution of code as its fallback mechanism. This approach has the advantage of being fast but it must be executed without any concurrency, which can hinder the performance. DMP-STM, conversely, uses STM as its fallback mechanism. Hence, with this solution the fallback path can achieve parallelism with STM transactions, although at a larger instrumentation overhead.

DMP-TM has been evaluated in an extensive study recurring to a synthetic benchmark composed by two disjoint hashmaps, one more prone to benefit transactions running HTM and another one more to benefit transactions running STM. DMP-TM demonstrated to be competitive when accesses are done only to one of the hashmaps, a test that is expected to be beneficial for respectively HTM and STM, depending on the hashmap accessed. Also DMP-TM demonstrated to achieve average performance gains of 1.5x (and up to 4x) in comparison to TinySTM whenever the larger hashmap is used with probability less than 10%. As for HTM, throughput gains in these workloads average 1.33x with peak gains that extend up to 4x. Finally, when compared with Hybrid NOrec, DMP-TM achieves throughput gains up to 10x. When considering workloads that are designed to be optimally managed by either STM or HTM, the proposed solution remains very competitive, achieving indistinguishable performance when compared to HTM, and averages overheads of 20 (and maximum of 30) vs TinySTM.

It should be noted that although there are a number of workload scenarios where DMP-TM outperforms all existing solutions, there are still workloads, in which memory accesses of transactions are not easily partitionable, in which the performances of the proposed solution does pay a performance toll vs state of the art Hybrid TMs. This is arguably not so surprising, given that existing literature in the area of TM has already proved that no-one-size fits all solution exists capable of delivering optimal performance across all possible workloads [9; 12; 13; 37]. Also, it should be noted that, it would be relatively straightforward to use adaptive techniques to enable/disable the proposed memory partitioning scheme [45], in case the application’s workload is unfavourable, and to resort to a conventional Hybrid TM approach (like Hybrid NOrec).

This way, we proved that Hybrid TM is still a promising concept and that for workloads characterized by accesses performed on disjoint pages, Hybrid TM proves to be better than HTM and STM by using the best of both implementations.

5.2 Future Work

For future work, it would be interesting to develop alternative implementations of the Contention Manager module, which may base their decisions on more sophisticated heuristics measured at runtime. One additional area of work that would be worth pursuing is on the development of a more realistic/complete memory allocator, as the current solution served just as a proof of concept aimed to support the execution of non-synthetic benchmarks (but, for instance, does not support free() operations).

It would be interesting to test DMP-TM with additional STMs, besides TinySTM, like, for instance, NOrec. As we have seen, in fact, in certain workloads (see Section 4.2.2) NOrec can outperform
TinySTM. Hence, by incorporating NOrec, as its STM backend, DMP-TM could further enhance its efficiency in these workloads. One interesting research direction is to extend DMP-TM to support heterogeneous STM backends, which would allow to run, for instance, TinySTM and NORec on different memory partitions.

Finally, with the upcoming of Intel Memory Protection Keys (MPK). It will be possible to change the memory protection of a partition without the current overhead associated. Currently, in fact, changes of the memory protection require invalidating translation lookaside buffer (TLB) entries across the entire system, and changing the protection of the memory in a memory region can imply changing the page-table entries for thousands (or more) pages. With MPK, instead, once the protection keys are set, a region of memory can be enabled or disabled with a single register write. For any application that frequently changes the protections on regions of its address space, the performance improvement will be large.


11. C. Click, “Azul’s experiences with hardware transactional memory,” in HP Labs’ Bay Area Workshop on Transactional Memory, 2009.


