Marcio Machado Pereira

“Scheduling and Serialization Techniques for Transactional Memories”

“Técnicas de Escalonamento e Serialização para Memórias Transacionais”

CAMPINAS
2015

University of Campinas
Institute of Computing

Universidade Estadual de Campinas
Instituto de Computação

Marcio Machado Pereira

"Scheduling and Serialization Techniques for Transactional Memories"

Supervisor(s)/Orientador(es)
Prof. Dr. Guido Costa Souza de Araújo
Prof. Dr. José Nelson Amaral

"Técnicas de Escalonamento e Serialização para Memórias Transacionais"

PhD Thesis presented to the Post Graduate Program of the Institute of Computing of the University of Campinas to obtain a PhD degree in Computer Science.

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Doutor em Ciência da Computação.

This volume corresponds to the version of the Thesis submitted to examining board by Marcio Machado Pereira, under the supervision of Prof. Dr. Guido Costa Souza de Araújo.

Este exemplar corresponde à versão da Tese apresentada à banca examinadora por Marcio Machado Pereira, sob orientação de Prof. Dr. Guido Costa Souza de Araújo.

[illegible]

Supervisor's signature

[illegible]

Assinatura do Orientador(a)

CAMPINAS
2015

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

P414s

Pereira, Marcio Machado, 1959-
Scheduling and serialization techniques for transactional memories / Marcio Machado Pereira. – Campinas, SP : [s.n.], 2015.

Orientador: Guido Costa Souza de Araújo.
Coorientador: José Nelson Amaral.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Memória transacional. 2. Programação paralela (Computação). 3. Processamento paralelo (Computadores). 4. Programação (Computadores). I. Araújo, Guido Costa Souza de,1962-. II. Amaral, José Nelson. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Técnicas de escalonamento e serialização para memórias transacionais
Palavras-chave em inglês:
Transactional memory
Parallel programming (Computer science)
Parallel processing (Electronic computers)
Computer programming
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
André Rauber Du Bois
Alexandro José Baldassin
Márcio Bastos Castro
Sandro Rigo
Data de defesa: 02-03-2015
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Defesa de Tese de Doutorado em Ciência da Computação, apresentada pelo(a) Doutorando(a) **Marcio Machado Pereira**, aprovado(a) em **02 de março de 2015**, pela Banca examinadora composta pelos Professores Doutores:

[illegible]

Prof(a). Dr(a). André Rauber Du Bois

Titular

[illegible]

Prof(a). Dr(a). Alexandro José Baldassin

Titular

[illegible]

Prof(a). Dr(a). Márcio Bastos Castro

Titular

[illegible]

Prof(a). Dr(a). Sandro Rigo

Titular

[illegible]

Prof(a). Dr(a). Guido Costa Souza de Araújo

Presidente(a)

Institute of Computing /*Instituto de Computação*
University of Campinas /*Universidade Estadual de Campinas*

Scheduling and Serialization Techniques for Transactional Memories

Marcio Machado Pereira[1]

March 02, 2015

[1]Financial support: CAPES scholarship 2011–2012, CNPq scholarship 2012–2013, CAPES (PDSE) scholarship 2014–2014

Abstract

In the last few years, Transactional Memories (TMs) have been shown to be a parallel programming model that can effectively combine performance improvement with ease of programming. Moreover, the recent introduction of (H)TM-based ISA extensions, by major microprocessor manufacturers, also seems to endorse TM as a programming model for today's parallel applications. One of the central issues in designing Software TM (STM) systems is to identify mechanisms or heuristics that can minimize contention arising from conflicting transactions. Although a number of mechanisms have been proposed to tackle contention, such techniques have a limited scope, because conflict is avoided by either interrupting or serializing transaction execution, thus considerably impacting performance. This work explores a complementary approach to boost the performance of STM through the use of schedulers. A TM scheduler is a software component that decides when a particular transaction should be executed. Their effectiveness is very sensitive to the accuracy of the metrics used to predict transaction behaviour, particularly in high-contention scenarios. This work proposes a new Dynamic Transaction Scheduler — DTS to select a transaction to execute next, based on a new policy that rewards success and an improved metric that measures the amount of effective work performed by a transaction.

Hardware TMs (HTM) are an interesting mechanism to implement TM as they integrate the support for transactions at the lowest, most efficient, architectural level. On the other hand, for some applications, HTMs can have their performance hindered by the lack of scalability and by limitations in cache store capacity. This work presents an extensive performance study of the implementation of HTM in the Haswell generation of Intel x86 core processors. It evaluates the strengths and weaknesses of this new architecture by exploring several dimensions in the space of TM application characteristics. This detailed performance study provides insights on the constraints imposed by the Intel's Transaction Synchronization Extension (Intel's TSX) and introduces a simple, but efficient, serialization policy for guaranteeing forward progress on top of the best-effort Intel's HTM which was critical to achieving performance.

Resumo

Nos últimos anos, Memórias Transacionais (*Transactional Memories* — TMs) têm-se mostrado um modelo de programação paralela que combina, de forma eficaz, a melhoria de desempenho com a facilidade de programação. Além disso, a recente introdução de extensões para suporte a TM por grandes fabricantes de microprocessadores, também parece endossá-la como um modelo de programação para aplicações paralelas. Uma das questões centrais na concepção de sistemas de TM em *Software* (STM) é identificar mecanismos ou heurísticas que possam minimizar a contenção decorrente dos conflitos entre transações. Apesar de já terem sido propostos vários mecanismos para reduzir a contenção, essas técnicas têm um alcance limitado, uma vez que o conflito é evitado por interrupção ou serialização da execução da transação, impactando consideravelmente o desempenho do programa.

Este trabalho explora uma abordagem complementar para melhorar o desempenho de STM através da utilização de escalonadores. Um escalonador de TM é um componente de *software* que decide quando uma determinada transação deve ser executada ou não. Sua eficácia é muito sensível às métricas usadas para prever o comportamento das transações, especialmente em cenários de alta contenção. Este trabalho propõe um novo escalonador, *Dynamic Transaction Scheduler* — DTS, para selecionar a próxima transação a ser executada. DTS é baseada em uma política de "recompensa pelo sucesso" e utiliza uma métrica que mede com melhor precisão o trabalho realizado por uma transação.

Memórias Transacionais em *Hardware* (HTMs) são mecanismos interessantes para implementar TM porque integram o suporte a transações no nível da arquitetura. Por outro lado, aplicações que usam HTM podem ter o seu desempenho dificultado pela falta de escalabilidade e transbordamento da *cache* de dados. Este trabalho apresenta um extenso estudo de desempenho de aplicações que usam HTM na arquitetura Haswell da Intel. Ele avalia os pontos fortes e fracos desta nova arquitetura, realizando uma exploração das várias características das aplicações de TM. Este estudo detalhado revela as restrições impostas pela nova arquitetura e introduz uma política de serialização simples, porém eficaz, para garantir o progresso das transações, além de proporcionar melhor desempenho.

To my wife, Lia

Acknowledgements

I would like to thank my committee members who were more than generous with their expertise and precious time. A special thanks to my supervisors Prof. Dr. J. Nelson Amaral and Prof. Dr. Guido Araújo for their excellent guidance, countless hours of reflecting, reading, encouraging, and most of all patience throughout the entire process. I would like to acknowledge and thank my colleague Matthew Gaudet who provided valuable feedback and made the completion of this research an enjoyable experience.

I thank all the faculty members, staff and students of the Computer Science Institute at UNICAMP as well as of the Department of Computing Science at the University of Alberta, whose presence and spirit have made both fantastic places to learn, to think, and to have fun.

Finally, and most importantly, I would like to thank my family. They were always supporting me and encouraging me with their best wishes. Most especially, I would like to thank my wife and best friend, Lia, for her love and tremendous support in each step of the way. No amount of words could fully convey my gratitude to her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The introduction of multicore architectures has renewed the search for programming models that can simplify the creation of efficient parallel applications. Among the various models proposed so far, Transactional Memory (TM) [29] is expected to enable parallel programming at lower programming complexity, while delivering improved performance over traditional lock-based systems. However, there are certain situations where transactional memory systems could actually perform worse. TM systems can outperform locks only when the executing workloads contain enough parallelism. When the workload lacks inherent parallelism, launching excessive transactions can adversely degrade performance. One of the central issues in designing Software TM (STM) systems is to identify mechanisms or heuristics that can minimize the contention arising from conflicting transactions. Although a number of mechanisms has been proposed to tackle contention [3, 16, 48], such techniques have a pessimistic approach, with respect to conflicting transactions, as they either interrupt or serialize their execution, thus impacting performance [17].

Hardware TMs (HTM) are an interesting option to implement TM because they integrate the support for transactions at the lowest, most efficient, architectural level. On the other hand, for some applications, HTMs can have their performance hindered by the lack of scalability and cache capacity overflow [44]. Software Transactional Memories (STMs) [26], in contrast to HTMs, tread in the opposite design direction. STMs are implemented atop the processor and, thus, allow more flexible designs as they do not limit the size of their critical data structures to the hardware constraints. Eventually, as research evolves, the combination of HTM and STM systems could lead to the best solution for a larger range of applications.

This thesis is organized as follows. Section 1.1 details the problem statement and Section 1.2 describes the contributions to improve the design of STMs, the recommendations on how to best use the new Intel HTM support as well the contributions on forward-progress policies to this architecture. Chapter 2 provides background on STM (Section 2.1) and HTM (Section 2.2), and introduces the Intel's Transactional Synchronization Extensions — TSX, for HTM support, with a brief description of transactional abort's causes and the various sources of data conflicts. The thesis is then divided into two parts: The first part, described in Chapter 3, details our contributions to STM and the second part, described in Chapters 4 and 5, details our contributions to HTM. On the first part, Section 3.1 proposes *Dynamic Transaction Scheduler* — DTS, a new scheduler to decide which transaction to execute next. Then, Section 3.2 introduces *Best Alternative Transaction* — BAT, a heuristic that guided the scheduler to switch a conflicting transaction by another with a lower conflicting probability; it defines *Percentage of Effective Work* — PEW, a new metric that measures the progression of each transaction, and presents the *Success-Rewarding Policy* — SRP, which is based on PEW and can be used by a scheduler to guide the scheduler choices. Following, Section 3.3 explains the experimental infrastructure and methodology for BAT and SRP and analyzes the proposed heuristics, while comparing then to other similar approaches from the literature (e.g., ATS [48] and Shrink [17]). Section 3.4 discusses related work of the first part. On the second part of the thesis, Section 4.1 describes the policies generally used on the evaluations of HTMs, and also introduces *SerControl*, a simple and effective serialization policy that is more efficient for TM on Intel's TSX than policies used so far [47, 45]. Section 4.2 describes our experimental setup, a detailed assessment of Intel's TSX and recommendations on how to best use the new Intel HTM support. Section 4.2.4 presents an evaluation of Intel's TSX using the STAMP benchmark suite [36] and Section 4.3 discusses related work. Finally, Chapter 5 explores, with the aid of a new tool called htm-*pBuilder*, the performance of Intel's TSX for various fall-back policy tunings and transaction properties; it also discusses future work. We conclude the work of the thesis on Chapter 6.

Some of the material used in this thesis has been published in the following papers:

Pereira M. M., Baldassin A., Araujo G., Buzato L. E. Transaction Scheduling using Conflict Avoidance and Contention Intensity. *In 20th International Conference on High Performance Computing (HiPC), 2013* DOI: 10.1109/HiPC.2013.6799126 Publication Year: 2013, Page(s): 236 - 245

Marcio Pereira, José Nelson Amaral and Guido Araújo. Measuring Effective Work to Reward Success in Dynamic Transaction Scheduling. *In 43nd International Conference On Parallel Processing (ICPP), 2014* September 9-12, 2014, Minneapolis, MN, USA

Marcio Machado Pereira, Matthew Gaudet, José Nelson Amaral and Guido Araujo.

Multi-dimensional Evaluation of Haswell's Transactional Memory Performance. *In 26th International Symposium On Computer Architecture And High Performance Computing (SBAC-PAD), 2014 IEEE* ; October 22-24, 2014 University Pierre et Marie Curie, Paris, France

Marcio Machado Pereira, Matthew Gaudet, José Nelson Amaral and Guido Araujo. Study of Hardware Transactional Memory Characteristics and Serialization Policies on Haswell. *Extended paper submitted to the Special Issue of Parallel Computer Journal* (Ref. No.: PARCO-D-14-00235) as one of the best papers of SBAC-PAD 2014.

1.1 Problem Statement

Although TM systems improve performance, mainly in the presence of a good amount of parallelism, they may considerably degrade performance for those workloads with high data contention rates. Contention occurs when more than one transaction access the same memory location, and at least one transaction writes to that location, causing a conflict. When a conflict occurs, the system has to decide which action to take.

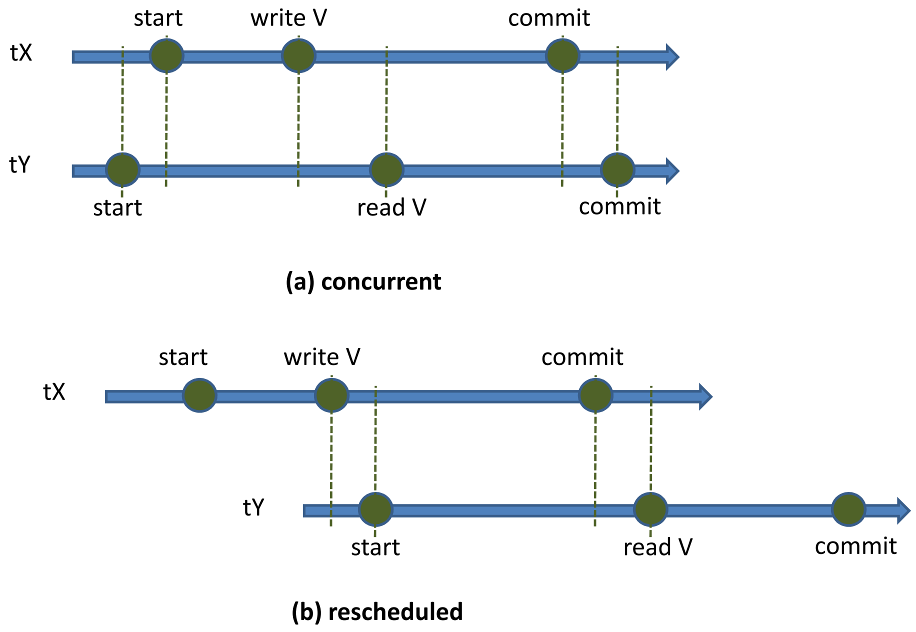
For instance, consider two transactions tX and tY referencing the same memory location m in an ETL–WB[1] STM implementation. If tX writes to m and later tY reads from m, a conflict occurs as illustrated in Figure 1.1(a). In order to deal with such conflict, a contention manager typically aborts tY and restarts it later.

Deciding which transaction to abort is part of a set of contention policies used by the Contention Manager — CM [23]. Most contention policies select to abort the transaction that produces the smallest impact in performance when a conflict occurs [43]. In general, managers cannot avoid conflicts because they can only select which transaction to abort and the amount of time to delay the restart of the aborted transaction. The benefits of delaying the restart of a transaction are not simple to anticipate. If an aborted transaction restarts too early it may cause new conflicts. If it restarts too late it may preclude the exploitation of program concurrency to its fullest. Such limitation makes it harder to effectively increase transaction throughput because managers can only act after a conflict has been detected.

Scheduling-based research [7, 17] have employed heuristics to predict a conflict before transaction scheduling. For instance, the conflict history of every transaction pair can be recorded and used to predict future conflicts [16]. Returning to the example of Figure 1.1,

[1]ETL–WB is a time-based encounter-time locking system where locks are acquired at the time data are written with a write-back update strategy that buffers writes until commit time.

Figure 1.1: Concurrent and ordered scheduling alternatives.



tY will not be allowed to start before tX commit, an approach called serialization. An important research question is whether serialization is the best solution. There are circumstances where some overlapping is possible, but producing such scheduling is far from simple. Consider the scheduling shown in Figure 1.1(b). Transaction tY reads variable V immediately after tX commits and, assuming no other dependencies in the code, this scheduling is the best attainable. However, arranging the transaction tY to start exactly at that point is not an easy task and might involve some instrumentation by the STM system which, in turn, might add a prohibitive overhead.

Approaches like ATS [48] and Shrink [17] do not offer any way to control the real scheduling of transactions other than yielding the processor and hoping that the operating system schedules the right thread with the right transaction. CAR-STM explicitly moves threads between the queues in each core [16]. However, it expects applications to provide a conflict probability (CP) for each transaction. The CP is generally difficult to determine because programs have multiple execution phases and the CP may vary over time.

Two research questions addressed in this work are (1) whether a TM scheduler that aims to avoid TM conflicts could anticipate that a transaction will abort and delay its dispatch, thus simulating the situation shown on Figure 1.1(b); and (2) what policies should be used to guide such a scheduler. The main idea is that a scheduler that has more information about the transactions' past behaviour is in a better position to select which speculative thread to schedule next than the operating system scheduler.

1.2 Contributions

This work makes four contributions to improve the design of STMs. First, it introduces DTS which implements a fully cooperative transaction scheduler. Second, it defines BAT heuristic, a proactive heuristic that, combined with DTS, improves STM performance on high contention scenarios. Third, it introduces PEW, a new metric that measures the progression of each transaction, and fourth, it presents SRP, a policy which is based on PEW and can be used by a scheduler to guide the scheduler choices. The new PEW metric can also be used by Contention Managers to improve the resolution of conflicts. [2]

This work also presents a performance evaluation of the HTM capabilities in Intel's Haswell micro-architecture called the Transactional Synchronization Extensions — TSX [33]. The goal is to study performance from the application perspective, providing a precise evaluation of the strengths and weaknesses of this architectural feature. To efficiently exploit the parallelism available through Intel's TSX, it is important to know the constraints imposed on software by its hardware design, and the requirements that must be fulfilled by software support systems.

This study also introduces *SerControl*, a simple and effective serialization policy that is more efficient for TM on Intel's TSX than policies used so far [47, 45] and a new tool — htm-*pBuilder* , that acts as a wrapper over Eigenbench [30] to allow fall-back policy tunings. We are currently integrating the proposed *SerControl* policy within the `libitm`, the TM library of the GCC compiler [1]. Through this integration it will be possible to achieve an abstraction capable of alleviating the task of developers to understand and deal with hardware details, through a much more user-friendly interface between the machine and the TM developer.

Research in TM is very promising because of the great power and simplicity of its programming model and its performance potential. Thus, this area has attracted increasing interest in recent years, and our contribution has focused on identifying the sorts of heuristics or techniques that can improve TM applications.

[2]Contention Managers with PEW is not explored in this work.

Chapter 2

Background Information

Transactional Memory, a lock-free alternative for shared state concurrency, makes it easier for programmers to develop parallel programs. With TM, programmers enclose a group of instructions within a transaction to execute the instructions in an atomic and isolated way.

Transactions appear as single operations that do not yield inconsistent state while running but not having committed or aborted yet. They also do not interfere with other running transactions and their outcome is always persisted. As a consequence of these properties, namely, atomicity [1], consistency [2] and isolation [3], transactions are serializable. The outcome of transactions can be reproduced by an equivalent sequential execution of seemingly atomic operations. Concurrency control for transactions can either be pessimistic or optimistic. Pessimistic concurrency control forces conservative locking of resources and results in low transaction throughput. Optimistic concurrency control delays the integrity checks of a transaction to its end. In case of a conflict, the transaction is aborted and gets restarted. When transactions are not long-running and do not conflict too often, optimistic concurrency control provides a very good performance with a negligible overhead of retries.

The concept of TM can be implemented in various ways. HTM provides an hardware implementation of TM, that extends CPU architectures by transactional components

[1] A transaction is atomic if the changes it makes to storage appear to happen indivisibly: either all the changes must be seen at once, or none of the changes can be seen.

[2] The system remains consistent after the execution of a transaction. If a transaction would leave the system in an inconsistent state, then the transaction is not allowed to complete.

[3] Transactional isolation is the inability of any component of the system to see partial results from an in-progress transaction. Isolation is a key part of proving consistency because, without isolation, invalid results could be produced by using invalid temporary results produced during the execution of a transaction.

such as transactional caches and an extended instruction set. All existing transactional memory systems are ‘best-effort’ because they do not guarantee speculative completion of all transactions due to capacity and functional limitations. Section 2.2 describes HTM implementations in more detail.

While TM was initially specified as hardware-supported lock-free data structures, much research has since gone into STM. STM does not require any hardware changes [4] and supports transaction handling entirely in software. Section 2.1 focus on STM systems.

2.1 Software Transactional Memory

Optimistic concurrency control is preferred by existing STM implementations. Once a transaction has been started at runtime, the underlying implementation starts to keep a read set and a write set. Both sets contain all variables and states that the transaction has read or altered. This is necessary for a later integrity check before committing. Also, as long as the transaction is pending, changes are not applied to the actual share variables, but on thread-local copies, often in form of a transaction log. Once the transaction has been verified as not conflicting, all of its changes are then flushed to the actual shared states in an atomic step. While this often contains some forms of locking, this behaviour is entirely transparent for the developer. In order to detect conflicting transactions, the STM implementation compares the read and write sets of a transaction with the actual states before committing. When another transaction has already altered a state and has committed successfully, the STM detects the discrepancy and aborts the transaction. Instead, the old read and write sets get discarded and refreshed, and the transactions restarts. To some extent, starvation situations can still occur, especially when a long-running transaction is steadily outpaced by other transactions that successfully commit first. Apart from that, STM provides mutual exclusion without explicit locking, and without the danger of the locking issues so far.

There are important limitations of STM. As TM transactions are limited to memory operations, they can only be used when coordinating access to shared state, but not to external resources. Furthermore, the transactional character requires operations to be irrevocable, so transactions must not have any side effects apart from modifications of shared state. For example, the usage of I/O operations inside transactions is disallowed. Furthermore, the length of transactions and the ratio of conflicting transactions have a lasting effect on the performance of STM deployments. The longer transactions take to execute, the more likely they cause conflicts and must be aborted, at least in case of many

[4] Actually, STM requires minimal hardware support, typically an atomic compare and swap operation, or equivalent.

contending transactions. When implemented with optimistic concurrency control, STM provides reasonable performance, as long as there are not many concurrent and conflicting write operations.

Initial attempts to mitigate the negative effects of conflicts relied on the so-called *contention management* — CM, introduced by Herlihy *et al.* in 2003 as a mean to guarantee progress in obstruction-free STM implementations [28]. The idea is very simple and elegant: when a transaction discovers a conflict, it consults a contention manager to determine how to proceed (e.g., restart immediately or restart after some delay). Despite significant progress on the design of contention managers, STM systems have not always been able to anticipate conflicts and increase system throughput. On the contrary, traditional contention managers use a reactive strategy, instead of focusing on avoiding conflicts.

Scheduling-based contention management approaches have emerged to counter the limitation exposed in CM [48, 16, 3, 17, 6, 35]. In special, the work by Attiya and Milani [6] provides a formal description of schedulers targeted at read-dominated workloads. The basic idea of this new strategy is to use some information about the past history to decide whether a given transaction should be scheduled for execution or not.

Adaptive Transaction Scheduling — ATS [48] is a transactional scheduler that keeps a per-thread measurement of contention intensity as a predictor to decide if a transaction should be allowed to start. In case the contention is too high, transactions are inserted into a queue and serialized, appropriately avoiding repeated aborts. ATS is very attractive since it lends itself to simple implementations.

CAR-STM [16] is a transactional scheduler that maintains a queue of transactions for each core. When a conflict between two transactions arises, the scheduler inserts the aborted transaction into the aborter's queue. Therefore, the likelihood of repeated aborts is decreased since the aborted transaction is serialized after the aborter.

Shrink [17] makes use of past access patterns of a thread to predict whether a transaction should be allowed to start in the future, thus preventing conflicts if the prediction is accurate. When a transaction is about to start, Shrink checks if any address in the predicted read and write sets is being written by an active transaction. If that is the case, the transaction is serialized. To avoid unnecessary serialization, Shrink also measures the contention intensity and serializes only if it is above a certain threshold.

Both ATS and Shrink force two potentially conflicting transactions to execute in sequence but do allow them to run concurrently with other non-conflicting transactions. The serialization is their main contention management mechanism. They keep track of the likelihood of a transaction to abort and, when its conflict probability reaches a given threshold, they serialize such transaction.

The Lightweight User-level Transaction Scheduler — LUTS is based on the assumption that it is better to schedule a different transaction instead of risking serializing the execution of a transaction with high conflict probability [38]. LUTS implements a fully cooperative scheduler, and does not rely on the system-level scheduler for the context switching of speculative threads. LUTS creates exactly as many system-level threads as the number of available processors and then manages the allocation of transactions to these threads. Having more information about the potential for conflicts, a scheduler such as LUTS is better able to do this thread allocation than the operating system. LUTS uses the concept of contention intensity, proposed in ATS, as a metric to measure transaction conflict. However, different from ATS, which is restricted to serializing or yielding the transaction, the LUTS heuristic finds a more appropriate transaction — with lower contention intensity — to schedule.

2.2 Hardware Transactional Memory

A Hardware Transactional Memory system uses dedicated hardware to accelerate transactional execution [29, 37, 12]. The HTM system starts a transaction by executing a register checkpoint with shadow register files. Whenever the transaction writes to memory, the transactional value produced by the write is stored separately from the original value by either buffering the transactional data in hardware buffers, such as the cache, or by logging the old value, a process called *data versioning*. Data versioning can be implemented by augmenting the cache with additional bits [25] or by using separate hardware structures, such as Bloom filters [46], to record the memory addresses read and written by the transaction. A conflict between two transactions is detected by comparing the read sets and the write sets of both transactions. If a conflict is detected, one of the transactions is rolled back by discarding its transactional writes, restoring the values in the registers to the values saved at the checkpoint at the start of the transaction, and discarding any transactional changes to the state of the program. When there is no conflict, the transaction commits the transactional data and discards both the transactional metadata and the values saved at the register checkpoint.

Software support for HTM systems is limited by the information and control provided by the hardware-software interface. Typically HTMs offer limited scope to implement contention management, used to provide forward progress in STM systems, and so must rely on a lock based fallback policy to provide forward-progress guarantees. For instance, the TM runtime in the IBM BG/Q machine adopts a policy of performing a certain number of retries for an aborting transaction before causing the transaction to serialize through the acquisition of a global lock [44].

Intel Transactional Synchronizations Extensions — TSX [33] is a recent addition to the Intel architecture that provides programmers with hardware transactional memory in the Haswell processor (Haswell processor characteristics are described in Appendix B). Intel's TSX provides two software interfaces to programmers: **Hardware Lock Elision** — HLE, a legacy-compatible instruction set extension, and **Restricted Transactional Memory** — RTM, a new instruction-set interface. HLE provides a prefixed instruction that indicates that a lock acquisition is to be elided, with the body executed in a transaction instead. If the transactional execution fails, the execution falls back to the original lock. The RTM interface executes code in a transaction, but provides no guarantee that a transactional execution will eventually commit. Therefore the program must always provide code to handle a transactional abort that can either restart the transaction or take a non-transactional path.

A processor can perform a transactional abort for numerous reasons. A primary cause is conflicting data accesses between transactions executing in different logical processors. Such conflicting accesses force an abort to ensure the preservation of transactional isolation. Transactional aborts may also occur because of limited transactional resources. For example, the amount of data accessed in a transactional region may exceed the HTM capacity limit. Intel's TSX uses the EAX register to communicate abort status to software. Causes for abort in TSX include execution overlap of transactional and non-transactional regions and system events such as system calls and page faults. During startup transactional programs experience a higher rate of aborts due to page faults. They experience a lower rate of such aborts after reaching steady state. However, in programs with very short run times, page-fault-induced aborts may appear to dominate.[5] A high rate of page-fault-induced aborts is also observed soon after large regions of memory are allocated.

[5]This type of abort affects most of the experiments with current benchmarks.

Chapter 3

STM Scheduling Policies

We have proposed DTS, in the same direction of the LUTS, a fully cooperative transaction scheduler presented by Nicácio [38], and also take same advantages when compared with other state-of-the-art TM schedulers [48, 7, 16]. First, both deal with the pseudo parallelism in an elegant way, by only spawning as many system-level threads as the number of available processor cores and handling the exceeding threads internally. Second, they allow the TM subsystem to efficiently access the runnable transaction queues and switch the execution to any of them. However, DTS is more flexible and scalable than LUTS because this was designed to work with multiple queues with priority. This allows the design of more precise proactive scheduling mechanisms, not possible with LUTS and also with other known approaches, that are restricted to either serialization or yielding.

3.1 Dynamic Transaction Scheduler

DTS is the component that implements policies to condition the execution of transactions with the sole purpose of increasing the probability of a concerted, frictionless, transactional execution. DTS is in sharp contrast with traditional contention management (CM), as CMs tend to focus primarily on conflict resolution. The differences between DTS and traditional CMs can be summarized as:

- CMs work at the moment a conflict among already active transactions is detected. DTS works before transactions are started.
- CMs are usually implemented inside the TM. DTS is implemented as a separate component that is logically at the same level as the STM, below the application and on top of the operating system (OS).

DTS acts proactively before transactions start. For a given set of transactions the main goal of DTS is to create a transaction execution schedule that prioritizes transactions that have done the most effective work. DTS associates a scheduling priority to each transactional thread. Thus, a scheduling decision takes into account the priority of all transactional threads in the program. Conceptually, DTS maintains a queue of runnable threads for each priority value. To determine which thread should run next, the scheduler looks for a non-empty queue with the highest priority and selects the thread at the head of the queue.

For a given queue of similar priorities, the DTS scheduling policy determines in which position in the queue its corresponding thread should be inserted. DTS uses a FIFO (First in, First Out) policy for the queue of threads. When a thread becomes runnable, it is inserted at the end of the queue corresponding to its priority. DTS provides a *sched_yield* interface, similar to the routine provided by UNIX-like operating systems, that forces the current thread to release the execution. The thread is inserted again at the end of the queue corresponding to its priority. No other events move a thread scheduled under the FIFO policy in the wait queue of runnable threads.

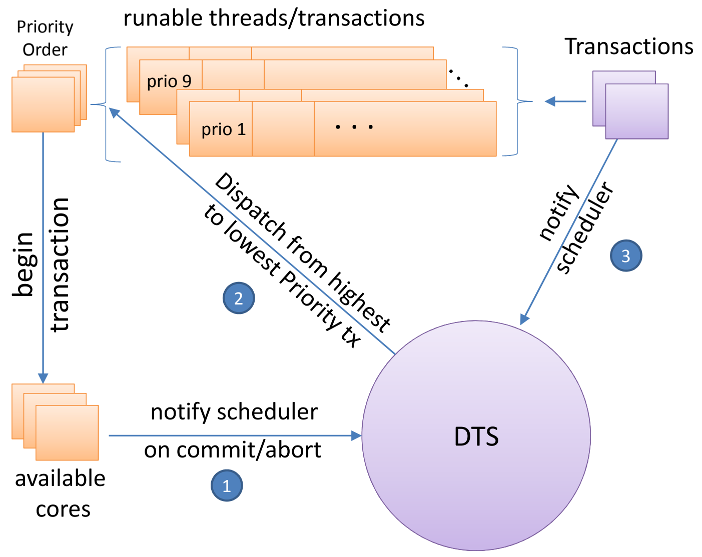
The number of queues used in DTS is a configuration parameter. For the prototype implementation working with the BAT heuristic (see Section 3.2.1) evaluated in this work, DTS is configured with just one queue and for the SRP policy (see Section 3.2.3), DTS is configured with ten queues with priority running from 1 to 10, respectively. In this configuration, the priority queue that a new transactional thread joins is determined by the PEW metric explained in Section 3.2.2.

Figure 3.1 shows a block diagram of the DTS scheduler. Every time a transactional thread commits or aborts, it notifies DTS ❶. DTS then calls the dispatcher to traverse the queues from highest to lowest priority and schedules transactions according to their priority ❷. This scheduling policy avoids starvation of transactions that abort repeatedly while trying to match the time delay between enemy transactions, which are competing for the same resources, as shown in Figure 1.1. When a transaction becomes ready to execute, it is encapsulated in a thread and the dispatcher enqueues the thread in the wait list of runnable threads according to its priority ❸.

3.2 Scheduling Heuristics

One of the central issues in designing STM systems is to identify mechanisms or heuristics that can minimize the contention arising from conflicting transactions. Although a number of mechanisms have been proposed to tackle contention, such techniques have a limited

Figure 3.1: DTS thread scheduling. The dispatcher selects from the highest to lowest priority transactional thread and maps them to available cores.


Priority Order
runable threads/transactions
prio 9
prio 1
Transactions
begin transaction
Dispatch from highest to lowest Priority tx
2
notify scheduler
3
notify scheduler on commit/abort
1
available cores
DTS

scope, as conflict is avoided by either interrupting or serializing transaction execution, thus considerably impacting performance. To deal with this limitation, we have proposed two new heuristics or policies that work with DTS. DTS implements a fully cooperative scheduler that switches a conflicting transaction by another one selected according to the heuristic premisses. The first one, called *Best Alternative Transaction* — BAT is described in Section 3.2.1 and the second one, called *Success-Reward Policy* — SRP, in Section 3.2.3. Experimental results, obtained using the STMBench7 [24] and STAMP benchmarks atop TinySTM [19, 18], show that the proposed heuristics produces better speedups when compared to other solutions.

3.2.1 Best Alternative Transaction

Usually, conflict avoidance mechanisms are implemented by continuously monitoring the level of transaction contention in the system and triggering the serialization of transactions while its value exceeds a given threshold [17]. Unfortunately, such techniques have a limited scope, as they do not try to replace the transaction whose execution has been delayed by another so as to sustain higher levels of transactional throughput. Instead, they avoid the conflict by simply serializing the execution of the potentially conflicting transactions.

In order to decrease contention, we have developed a new, proactive heuristic, named Best Alternative Transaction — BAT, which integrates the conflict avoidance mechanism proposed by Nicácio [38] with the *Contention Intensity* heuristic proposed by Yoo and Lee in the ATS system [48]. By doing so, we show that BAT effectively improves STM performance, mainly on high contention applications. Before introducing BAT, we will first outline the Contention Intensity — CI heuristic.

The rationale behind CI

The effectiveness of a transaction, that is, the chance that once started it progresses and commits, is directly related to the competition for shared resources encountered during its execution. In ATS, each transaction, materialized by a thread, maintains its own estimate of such competition, by measuring an index, called *Contention Intensity* — CI. Equation 3.1 below shows how ATS defines the contention intensity for the n-th activation of a transaction:

$$CI_n = \alpha * CI_{n-1} + (1 - \alpha) * CC \tag{3.1}$$

Contention intensity is computed by combining, with different weights, two components of the contention experienced by the transaction. The CI_{n-1} component captures the contention experienced by the transaction from its first ($n = 1$) activation up to its ($n - 1$)-th activation. The component CC measures the transaction contention, for its current (n-th) activation. A weight α is used to adjust the importance of the past and current contention components in the equation. Note that α and its complement are used in the computation of CI_n. By applying competitive learning between these two predictors, the α can be adjusted automatically. However, throughout our experiments, we fixed α to 0.30, giving a little more weight to CC, because this value showed the best performance. Initially, CI is set to 0, and the equation is evaluated at each commit or abort operation, with CC set to 0 on commits and to 1 on aborts.

When a transaction T is about to be started, the scheduler checks whether the CI of T is above a given threshold (again, throughout our experiments, we set the threshold at 0.70). If this is the case, the BAT heuristic is enabled to find an appropriate alternative transaction to take the place of T and a call to the method `sched_switch` is carried out to replace T with the newly selected transaction. By contrast, ATS simply serializes transactions whose CI has surpassed a given threshold. Of course, if the CI of T is below the threshold, then it is scheduled for execution (re-execution) without further delay.

The rationale behind BAT

BAT is based on a very simple rationale: maintain a global snapshot of the transactions active in the STM and select, as the next transaction to be executed, the transaction with the lowest probability of conflicting with those in the snapshot. The effectiveness of BAT depends on two factors:

- high accuracy: the transaction chosen by BAT should have a high probability of commit;
- low overhead: the execution of BAT must have a low impact on the overall execution of the transactional system.

The accuracy of BAT is highly dependent on the diversity of transactions available in the application. It is not difficult to see why by examining the extreme case where the application has only one transaction; the worst case scenario. At any moment, the STM will execute several instances of the same transaction. These, in their turn, will have the same structure, same expected duration and compete for the same shared resources. So, the probability of conflicts happening is going to be very high. In addition, at any moment, BAT will not be able to find a best alternative transaction to include in the set of active transactions because there is only one transaction available. In the best case scenario, BAT will be dealing with an application structured as many different transactions, possibly reading/writing different locations of the shared data space during their execution. Here, BAT is going to have at its disposal a diverse set of transactions from which it is going to pick the transaction the chance of conflict with the ones already in the snapshot, this is the best alternative transaction.

Suppose an activation and execution of BAT takes on average $\mathcal{C}$ cycles. The policy used to decide whether it is worth imposing the $\mathcal{C}$ overhead on the STM to minimize conflicts among transactions is the following: if the average duration of a transaction is shorter than $k \times \mathcal{C}$, where k has been found by actively monitoring the duration of transactions in the applications, then an heuristic for *short transactions* must be activated. Otherwise, the STM is considered as working mostly with *long transactions* and BAT for long transactions must be activated. In summary, the duration $k \times \mathcal{C}$ splits transactions into the *classes* short and long, and in each case BAT chooses an adequate heuristic to keep the overhead of DTS as low as possible. The factor k is computed using a sliding window average of the durations of the last *window* transactions with *window* set to one hundred (100); for the experiments the threshold for $k \times \mathcal{C}$ has been set to 100.000 cycles. During the profiling carried out to determine the threshold we have also found that an hysteresis should exist between the computation of the threshold and its use in the activation of each heuristic.

DTS switches to the heuristic adequate for *long transactions* only after two successive values of the sliding window average are above the threshold, and it is switched back to the heuristic for *short transactions* as soon as the average falls below the threshold. Next, we discuss the behaviour of DTS in each of these scenarios.

Scenario for short transactions

As said before, the heuristic chosen to select the next transaction to run in a short transactions scenario should have near-zero runtime overhead and a fair conflict prediction. So, in this case the best heuristic is to simply let DTS scheduler select as the next transaction to run the transaction in the FIFO queue different from the current transaction.

Scenario for long transactions

The goal of our heuristic in this scenario is to predict the best candidate transaction for execution given a set of active transactions. We say that a transaction is the best candidate when its conflict probability is the lowest one among the competing transactions. During execution time we record the past conflict history of a transaction in a table. In order to clarify the concepts of our higher-accuracy heuristic we provide now a detailed example:

Assume a system with 4 cores executing an application containing 3 transactions, represented by their unique identifiers (IDs) 1, 2 and 3. Besides the contention intensity (CI) maintained for each transaction, there are three more metadata employed by the higher-accuracy BAT heuristic: a vector with active transactions (`activeTx`), a conflict table (`conflictTable`), and a summary of the best candidate transactions (`bestTx`) as illustrated in Figure. 3.2. The `activeTx` vector maintains, for each core, the identification number of the transaction that it is currently running on that core. When no transaction is assigned to a core we make use of the ID -1. The active transactions in the system illustrated in Figure. 3.2 is given by the set {-1, 1, 3, 2}. Therefore, core 0 is not executing any transaction, whereas cores 1, 2, and 3 are executing transactions 1, 3, and 2, respectively.

Assume now that core 0 is about to start a transaction. In order to pick a transaction for execution, the algorithm accesses the `conflictTable` metadata. Each line in this table identifies a transaction set; each column quantifies the conflict probability of starting a specific transaction when that respective set is active. A hash function is responsible to map each active transactions set to a line index, as illustrated in Figure. 3.2. For this specific example, the conflict probability of starting transactions 1, 2, and 3 when the active set is {-1, 1, 3, 2} is 0.5, 0.2, and 0.1, respectively. Therefore, we conclude that core 0 should choose transaction 3, since it has the lowest conflict probability.

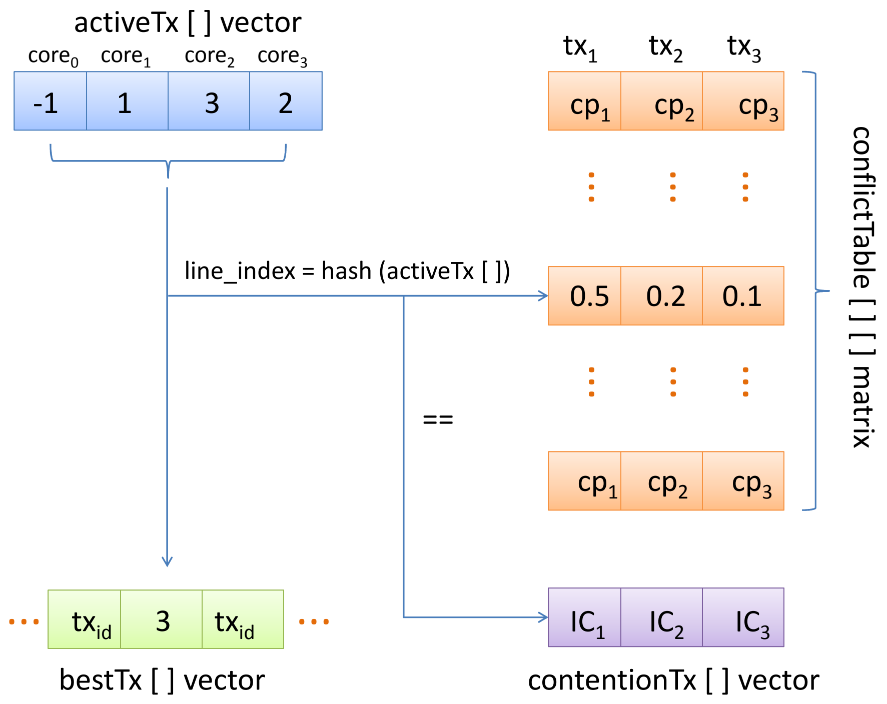


Figure 3.2: An example illustrating BAT heuristic metadata and how the best candidate transaction is selected for a given active transaction set.

Note that accessing the `conflictTable` every time a transaction is about to start may introduce unnecessary overhead. In order to avoid that we employ a summary vector. The same index returned by the hash function is used to access the `bestTx` vector (see Figure. 3.2), that stores the identifier of the best candidate transaction, in this case transaction 3. How this vector is updated is discussed below. For clarity, the pseudo-code for BAT is shown in Listing 4.1.

After a transaction is chosen, we invoke `sched_switch` and update `activeTx`, as described in the `start` operation (lines 9 to 13 of Listing 4.1). Now suppose that the selected transaction running on core 0 aborts after detecting a conflict (lines 15 to 29 of Listing 4.1). Initially, `activeTx` is updated to take into account that no transaction is running on this specific core anymore (line 16). Because there was a conflict, we need to increase the conflict probability of running this transaction with the active set of transactions (lines 17 to 19). Note that the active set is not necessarily the same one that was active when the transaction started. If we assume it is, then the old conflict probability (0.1) must be increased. Assume that the constant 0.1 is always added to the old value. In this scenario, the new conflict probability for transaction 3 must be increased to 0.2. Recall from the illustration given in Figure. 3.2 that the conflict probabilities for transactions 1 and 2 are, respectively, 0.5 and 0.2. Therefore, `bestTx` must be updated because there are other transactions with lower or equal conflict probability than transaction 3 (in the example, transaction 2). This action is performed by the lines 20 to 29 in the pseudo-code.

Listing 3.1: Moderate-Overhead High-Accuracy BAT heuristic

```
0  int activeTx[], bestTx[];
1  double contentionTx[];
2  double conflitcTable[][];

4  upon stm_init
5    resetBestTx();
6    resetCT();
7    for each core i activeTx[i] = INVALID;

9  upon start
10   int line_index = hash(activeTx);
11   int tx_id = bestTx[line_index];
12   sched_switch(tx_id);
13   updateActiveTx(thisCore, tx_id);

15 upon abort
16   updateActiveTx(thisCore, INVALID);
17   updateContentionIntensity(activeTx)
18   int line_index = hash(activeTx);
19   increaseProbCT(line_index, tx_id);
20   if (bestTx[line_index] == tx_id) {
21     for each transaction tx {
22       if (conflictTable[line_index][tx] <
23       conflictTable[line_index][tx_id])
24       or (conflictTable[line_index][tx] ==
25       conflictTable[line_index][tx_id])
26       and (contentionTx[tx] <= contentionTx[tx_id])
27     bestTx[line_index] = tx;
28      }
29   }

31 upon commit
32   updateActiveTx(thisCore, INVALID);
33   updateContentionIntensity(activeTx)
34   int line_index = hash(activeTx);
35   decreaseProbCT(line_index, tx_id);
36   if (conflictTable[line_index][tx_id] <
37       conflictTable[line_index][bestTx[line_index]])
```

```
38      or (conflictTable[line_index][tx_id] ==
39          conflictTable[line_index][bestTx[line_index]])
40      and (contentionTx[tx_id] <=
41           contentionTx[bestTx[line_index]])
42        bestTx[line_index] = tx_id;
```

We only need to check if a better transaction exists if the aborted transaction was previously in `bestTx` (line 20). In this case, we need to check every column of `conflictTable` in order to find the new best value (lines 21 to 28). When two transactions have the same conflict probability, as in the example, the algorithm makes use of the contention intensity of both transactions, as a tiebreaker. This strategy is based on the fact that the `conflitcTable` indicates only the probability of conflict for a specific scenario but does not give a more precise idea of conflict history of the transaction itself, better shown by its contention intensity.

If the selected transaction commits instead of aborting we need to decrease its conflict probability (pseudo code lines 31 to 42). This operation is similar to abort, except that we do not need to check every column of `conflictTable` to find a better candidate transaction. The only way for `bestTx` to change is if the committing transaction was not previously the best choice and, as a result of decreasing its conflict probability (line 35), it became the best choice (lines 36 to 42). For instance, if we picked transaction 2 instead of 3 during the start operation (assuming there was no transaction 3 to be dispatched) then, at commit time, we would update its conflict probability to 0.1 and update `bestTx` to reflect that transaction 2 maybe now the best candidate, just depending if its CI is less than or equal to the CI of the transaction 3.

Contrary to the low-overhead heuristic suitable for short transactions, where we might only use one variable by transaction to hold its contention intensity, this high-accuracy heuristic is more elaborated and, consequently, results in a higher runtime overhead. This is because such a heuristic can capture information about which transactions conflict with others. Once this information is collected it can be viewed in the context of global data structures named `conflictTable`, `activeTx` and `bestTx` in order to predict what is the best transaction to execute given a set of running transactions. BAT is constructed by combining these heuristics into a transaction predictor that can be dynamically controlled.

3.2.2 PEW Metric

Reducing the number of aborting transactions is an effective way to improve the efficiency of a TM system. Aborting a transaction can be costly. First, all the completed work done by an aborted transaction is wasted and has to be re-done. If an aborted transaction has aborted another transactions earlier, the total amount of work lost is even larger. Secondly, the cost to rolling back a transaction may require large memory footprint and bandwidth.

To prevent costly abortions, DTS uses the new Success-Rewarding Policy — SRPthat associates, to each transaction, a priority based on the Percentage of the Effective Work — PEW that the transaction has done so far. An *execution slice* is formed by k consecutive attempts to execute an static transaction t. For the experiments reported in this work $k = 20$. PEW records, at the end of the execution of each slice, $W_C(t, n)$ which is the sum of the machine cycles executed by a transaction t when it commits during slice n and $W_A(t, n)$ which is the sum of the machine cycles executed by t when it aborts during slice n. The total work done by transaction t during slice n is computed as follows:

$$W_T(t, n) \quad = \quad W_C(t, n) + W_A(t, n) \tag{3.2}$$

The computation of the metric keeps counters to measure the total/effective work done by t. These counters are defined as follows:

$$\begin{aligned} T(t, n) &= \alpha \times T(t, n-1) + (1-\alpha) \times W_T(t, n) \\ E(t, n) &= \alpha \times E(t, n-1) + (1-\alpha) \times W_C(t, n) \end{aligned} \tag{3.3}$$

where n is the current execution slice and α is a tuneable weight that controls the contribution of the past execution slices to the total/effective work (conversely, $1-\alpha$ is the contribution of slice n). $T(t, n)$ is the total amount of work executed by transaction t at the end of slice n and includes contributions from all previous execution slices, while $W_T(t, n)$ is the work done by transaction t *during* slice n. The work done by an execution of t is only effective when that execution commits. $E(t, n)$ is the total amount of *effective* work done by t at the end of slice n while $W_C(t, n)$ is the amount of effective work done during slice n.

The weighted percentage of effective work performed by t after n slices of executions is given by:

$$PEW(t, n) = \frac{E(t, n)}{T(t, n)} \tag{3.4}$$

The value of PEW can either be used to dynamically set the scheduling priority of the thread that is running the current transaction or to resolve conflicts by a conflict manager.

When PEW is evaluated, the priority of the current transaction is set to a range from 1 to 10, where 10 is the highest priority. Transactions with PEW between 0% and 10% belong to priority 1, 11% to 20% belongs to priority 2 and so on.

3.2.3 Success-Rewarding Policy

A careful analysis reveals that, under high-contention workloads, scheduling transactions only according to their PEW value may degrade performance because such strategy can lead to higher transaction conflict rates among transactions within the same priority queue. To deal with that, the following policy is proposed to reward success:

- Immediately after a transaction t aborts, compare the value of $PEW(t, n)$, where n is the current execution slice, against a *reward threshold.* If $PEW(t, n)$ is below this threshold, raise a *yield flag* for t to indicate that t should yield to other, more successful, transactions at the beginning of its next re-execution.

- If the *yield flag* of a transaction t, which is ready to execute, is raised, then clear the flag and place t at the end of its priority queue. Proceed fetching the next transaction that is ready to execute according to the priority mechanism described above.

The use of the reward threshold and the yield flag to make a transaction lose a turn once, after it aborts, changes the FIFO policy for the priority queues. Consider two transactions t_X and t_Y that conflict; t_Y gets aborted and restarts. The goal of the DTS is to delay the restart of t_Y by an appropriate amount of time, so that t_X and t_Y can still potentially overlap, but T_Y will not perform the first conflicting access until t_X commits. With the yield flag, t_Y misses one scheduling opportunity and DTS achieves this goal. Although very simple, this policy has shown to be effective on avoiding TM conflicts by delaying the transaction that has high probability to abort, thus leading to an affirmative answer to the first research question posed at the end of Section 1.1 as indicated by the experimental evaluation (see Section 3.3.3).

In general, programs tend to exhibit different execution phases which make it really difficult to devise a scheduling policy that works effectively all the time. Nevertheless, a careful experimental evaluation have indicated that the combination of DTS and SRP improves the STM performance for workloads with high data contention rates.

3.3 Experimental Evaluation

This section presents an experimental evaluation using a prototype implementation of DTS with BAT and SRP. The main findings from this evaluation are:

- Scheduling-based contention management approaches — such as DTS, LUTS and ATS — have the potential to improve the performance of TM applications because their influence in the execution of a transaction is not limited to a reaction to a conflict. However, early experimental results indicated that this approach is very sensitive to the metrics and policies used [38]. TM schedulers that either interrupt or serialize the execution of transactions may impact performance. For instance, excessive serialization of memory transactions, as occurs in ATS (see Figure 3.11), may result in too little concurrency. On the other hand, miss-predictions — for instance, due to Bloom filter's false-positive that occur in Shrink (see Figure 3.11) — serialize transactions unnecessarily and may lead to slowdown of programs.

- Among its competitors, DTS is the scheduler that most successfully explores the full potential of the new PEW metric and results in the best performance results. The TM scheduler proposed by Yoo and Lee [48] guided by the PEW metric does not result in great performance benefits because of serialization. DTS prevent transactions that are likely to cause a conflict from running even when there are enough CPUs to run them all. Therefore, when the number of dynamic transactions does not exceed the number of available processor cores (24 in our experiments), as shown in Section 3.3.1), all threads have a CPU to run, but applying the SRP policy (see Section 3.2.3), the scheduler ends up creating the situation in Figure 1.1(b) by delaying the transaction that has high probability to abort. With this strategy, the system retains fairness, only reducing potential conflicts and, consequently, the number of aborts.

- The BAT heuristic is expected to provide speedup gains when transaction diversity is reasonably large and contention is high. But similar to the heuristics in ATS, BAT only counts the number of aborts and commits but does not take into account the amount of work performed by the transactions. For instance, suppose that there are two transactions tX and tY that are queuing on the scheduler. Assume that tX has a history of ten aborts in the last twenty executions, while tY has a history of only five aborts in the last twenty executions. Hence tX should have a contention intensity (CI) that is higher than the CI for tY.[1] Predictors based on CI, would

[1]The value of contention intensity also depends on α and its past abort history. The weight α is used to adjust the importance of the past and current contention components in the CI equation [48].

> thus choose tY to be scheduled, instead of tX. Now assume that the ten aborts of tX amount to 20% of the *total work*, measured as the number of execution cycles, of tX while the five aborts of tY are responsible for 80% of the total work of tY. Therefore, even though tY had fewer aborts, it is wasting significantly more cycles, and a good predictor should select tX to be scheduled.

This insight led to the use of *effective work* as a better metric for the new scheduling policy presented in this work.

3.3.1 Experimental Infrastructure and Methodology

The experiments were conducted on a machine with two 6-core Intel Xeon E-5645 processors (24 cores in total, hyper-threading enabled), 32GB of RAM and clocked at 2.40GHz. The machine runs a standard Ubuntu Server 10.04.3 LTS amd64.

The experiments were based on the prototype of DTS built atop of TinySTM implementation, version 1.0.0, configured with the write-back and encounter-time locking — ETL strategy. The contention policy adopted was SUICIDE, which immediately restarts a transaction on abort. Other approaches, such as full lazy conflict detection [43] are likely to benefit from this strategy by yielding a more precise estimation of the rate of progress, but they are orthogonal to the strategy presented here.

This evaluation compares DTS with the prototype implementation of Yoo and Lee, Adaptive Transaction Scheduling — ATS [48], inside of TinySTM. ATS employs a single global queue for all transactions. The tuning of the ATS prototype included a sensibility study on α with values 0.3, 0.5 and 0.75, and on the CI threshold[2] with values 0.3, 0.5 and 0.7, which indicated that the combination of 0.75 for α and 0.7 for threshold yields the best overall performance, and thus was adopted in the experiments.

Another comparison was performed with an implementation of the Shrink scheduler proposed by Dragojevic *et al.* [17]. The code for Shrink, for TinySTM version 0.9.5, was taken from the authors website and adapted to the current version of TinySTM (1.0.0). The configuration parameters in the code are left unchanged.

The LUTS scheduler with the conflit-avoidance heuristic was used with the experiments aimed at comparing BAT heuristics. In order to predict the conflict beforehand LUTS was adopted, for the contention police, $\alpha = 0.75$ and threshold $= 0.5$.

All applications were compiled using the gcc compiler version 4.4.3. The results presented in next sessions are average over twenty executions and the graphs show a 95%

[2]A threshold on the Contention Intensity indicating whether a transaction should be serialized or not.

confidence interval — in some plots this interval is too small to be visible. All speedup results are in comparison to a non-thread-safe sequential execution, *i.e.* the baseline does not incur any transaction overhead.

Hardware performance counters, available on virtually all modern multiprocessors, were used to collect timing information to evaluate PEW. The number of attempts to execute a transaction that forms an execution slice for the PEW heuristic was empirically determined by running representative programs. Values considered include 10, 20, 50, 100 and 200 executions. The prototype evaluated in this work uses a slice of 20 executions because it produced the best overall results. Standard tuning techniques can be used to determine the appropriate slice size, for any TM system that use PEW.

For the experiments with the STAMP applications, the parameters listed in Table 3.1 were adopted [36]. They are the recommended configurations and data set for use in real machines. For the applications Vacation and Kmeans, only the results for the High configuration are shown because the results for the Low configuration are very similar.

Table 3.1: STAMP configurations

Application	Arguments
`bayes`	-v32 -r4096 -n10 -p40 -i2 -e8 -s1
`genome`	-g16384 -s64 -n16777216
`intruder`	-a10 -l128 -n262144 -s1
`kmeans`	-m15 -n15 -t0.00001 -i random-n65536-d32-c16
`labyrinth`	-i random-x512-y512-z7-n512
`ssca2`	-s20 -i1.0 -u1.0 -l3 -p3
`vacation`	-n4 -q60 -u90 -r1048576 -t4194304
`yada`	-a15 -i ttimeu1000000.2

The well-known and widely accepted STMBench7 [24] benchmark is used to evaluate DTS in high-contention workloads. STMBench7 that can create realistic transactional workloads corresponding to a wide variety of applications, such as CAD, CAM and CASE applications. STMBench7 uses a graph composed of objects (atomic and composite) as the basis for the implementation of over forty different transactional operations and thus provides a reasonably large transaction diversity. The configurations used for the STMBench7 are in Table 3.2:

Table 3.2: STMBench7 configuration

Attribute	Value
Long traversals	false
Workload types	read dominated, read-write
Duration	5000ms
Size	Small, Medium, Big and Huge

3.3.2 Evaluating BAT on High-Contention Workloads

DTS configured to working with the BAT heuristic is expected to provide speedup gains when transaction diversity is reasonably large and contention is high. Higher contention is obtained by increasing the number of threads, thus transactions, simultaneously active. To evaluate BAT, we choose STMBench7 because this benchmark uses a graph composed of objects (atomic and composite) as the basis for the implementation of over forty different transactional operations. Therefore, it is fair to say that this benchmark provides a reasonably large transaction diversity.

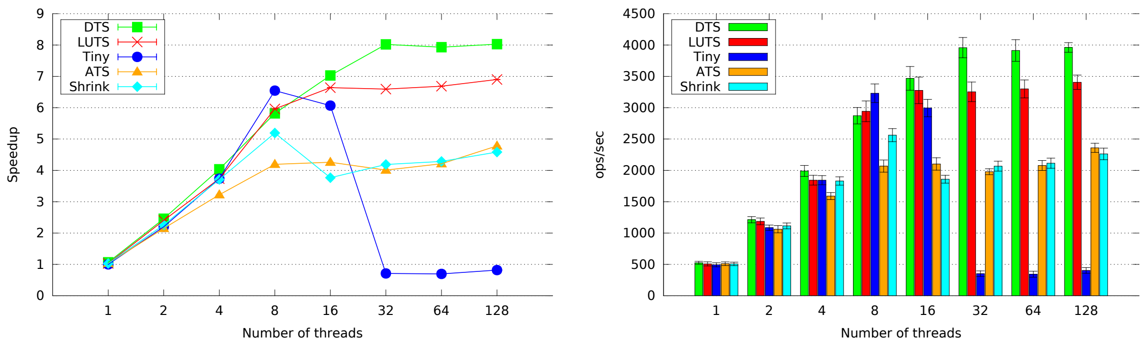
As shown in the graphs of Figures 3.3a, 3.3b, 3.3c and 3.3d, DTS performance is about the same as other systems in the interval that goes from 1 to 16 threads given that the number of threads is relatively small, generating moderate contention. This shows that DTS does not impose a significant overhead in workloads that do not favor the BAT heuristic.

The benefits of the BAT heuristic becomes clear as the number of threads as well the sizes of the data sets increase. For configurations Small (Figure 3.3d) and Medium (Figure 3.3c) DTS gives better speedups from 64 threads and up. For configurations Big (Figure 3.3b) and Huge (Figure 3.3a), there is an improvement over the other systems, as soon as the number of threads reaches 32.

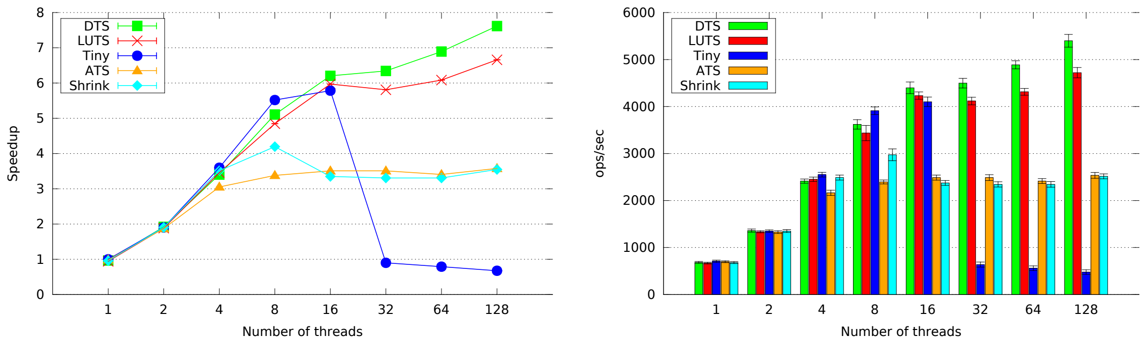
In the STMBench7 Big and Huge scenarios the length of the average transaction considerably grows (as the transaction length increases with the data size). This confirms that BAT selects a best transaction alternative for the cases when the cost of an abort is larger due to long running transactions.

Figure 3.3: STMBench7: DTS+BAT performance and speedup comparisons

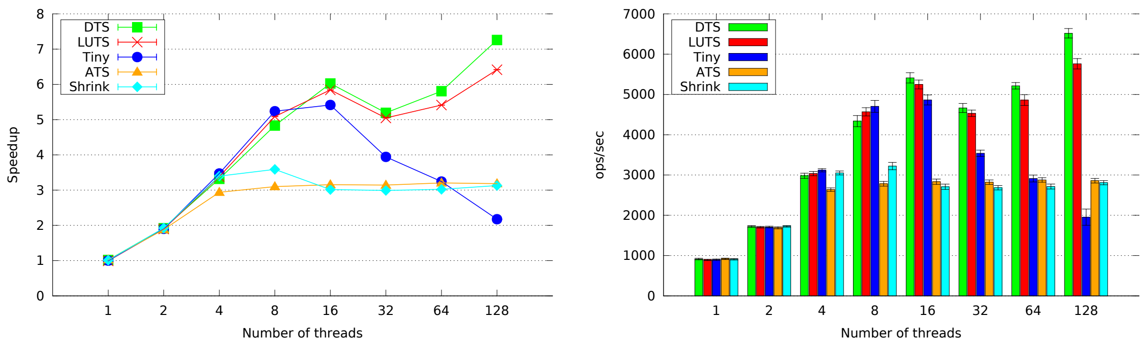
(a) Huge



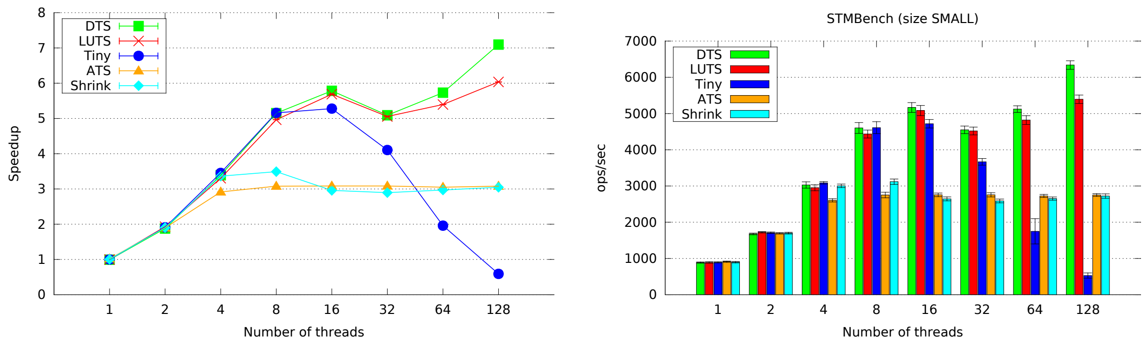
(b) Big



(c) Medium



(d) Small

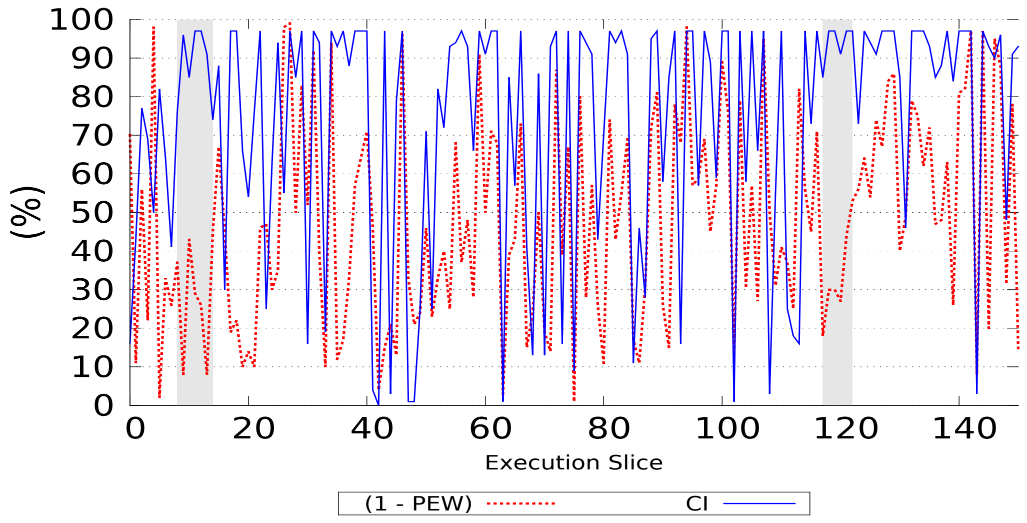


3.3.3 Comparing PEW with Contention Intensity

One of the claims in this work — which needs to be evaluated — is that the PEW metric is an improvement over the Contention Intensity metric used in heuristics elsewhere. To evaluate this claim TinySTM was instrumented to measure the effective work and the contention intensity every time a transaction starts, commits or aborts. This measurement used the original TinySTM scheduling and SUICIDE conflict resolution policie. The only interference is the overhead introduced to collect the data to compute both the PEW and CI metrics, which affects equally all transactions. While this data was collected for several applications, it is illustrated here only for a transaction from `yada`.

Yoo and Lee define the contention intensity (CI) as a dynamic average based on current available contention information [48] that is related to the percentage of aborts in relation to the total attempts to commit. PEW measures the percentage of work that was done by committed attempts in relation to the total work done by all executions of the transaction. For a visual comparison, Figure 3.4 shows (1 - PEW) which is the amount of wasted work (in red) and the level of CI (in blue) for 150 slices (3000 executions of the transaction).

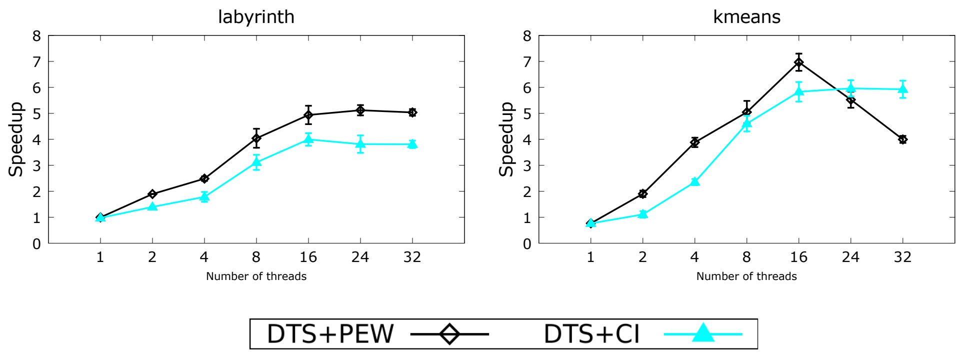
Figure 3.4: Percentage of wasted work ($1 - PEW$) and percentage of contention intensity for 150 execution slices of a single transaction in `yada`.



The first shaded area in the graph of Figure 3.4 illustrates that, without loss of accuracy, the percentage of aborts computed by contention intensity can be a poor indicator of the amount of work that the transaction has wasted. In that region the CI indicates that more than 75% of the transaction starts may have been aborted, but PEW indicates that less than 30% of the cycles executed by the transaction were wasted. Several other regions of the graph also indicate that, and thus, for this particular transaction, simply counting the number of aborts and commits over-estimates the resources wasted by the transaction in unsuccessful attempts to execute.

Comparisons such as the one exemplified by the graph in Figure 3.4 indicate that the PEW metric gives a more precise estimate of the success of a transaction than the CI metric. But, does this increased precision leads to better performance? Or perhaps the new scheduler, DTS, would perform just as well using the CI heuristic? A version of DTS guided by the CI metric was also created for this evaluation. Results of an experimental evaluation with the STAMP benchmark suite indicate that PEW is indeed responsible for some of the performance improvement. For instance, Figure 3.5 indicates that the use of PEW, as opposed to CI, is responsible for approximately 20% of the speedup of `labyrinth` and 14% of the speedup of `kmeans`. The performance drop of DTS+PEW in `kmeans` for more than 16 threads is explained by the high level of contention (see Table 3.1) and by the increase in conflict rates among transactions (see Section 3.2.3). SRP eliminates this performance drop as indicated by the experimental results in Section (3.3.3).

Figure 3.5: Comparing DTS with PEW metric and CI metric.

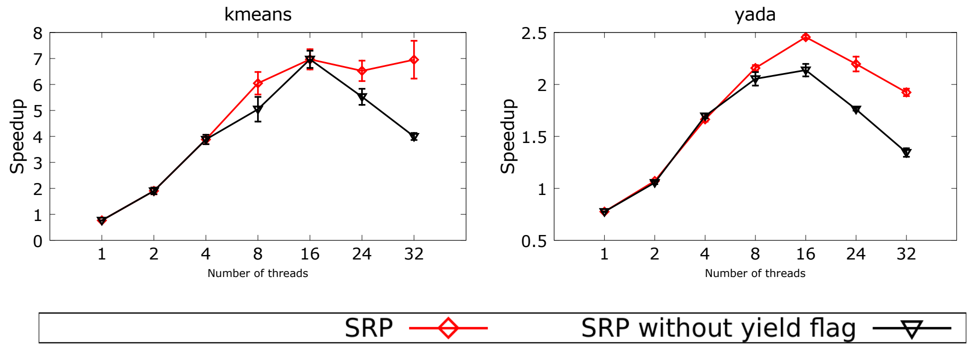


Is the Yield Flag Necessary?

Section 3.2.3 presented SRP that changes the policy of the priority queues through the use of the reward threshold and the yield flag. An important question for the evaluation of the SRP is whether the additional complexity of forcing a transaction to yield, after an abort, results in performance variations. For this evaluation, the *reward threshold* that produced the best overall results, determined empirically, was 50%. Other values considered were 30%, 70% and 100%.

Figure 3.6 shows the performance of DTS for `kmeans` and `yada` using the SRP police without the yield flag and the PEW metric. These results indicate that rewarding successful transactions by forcing the unsuccessful ones to yield tends to improve the performance for higher number of threads when contention is expected to be higher because more transactions are running simultaneously. In `kmeans`, the performance drop disappeared. `Labyrinth` did not show this effect and therefore is not shown here. Instead, the same effect is shown for `yada`.

Figure 3.6: Comparison of SRP and SRP without the yield flag, used in DTS.



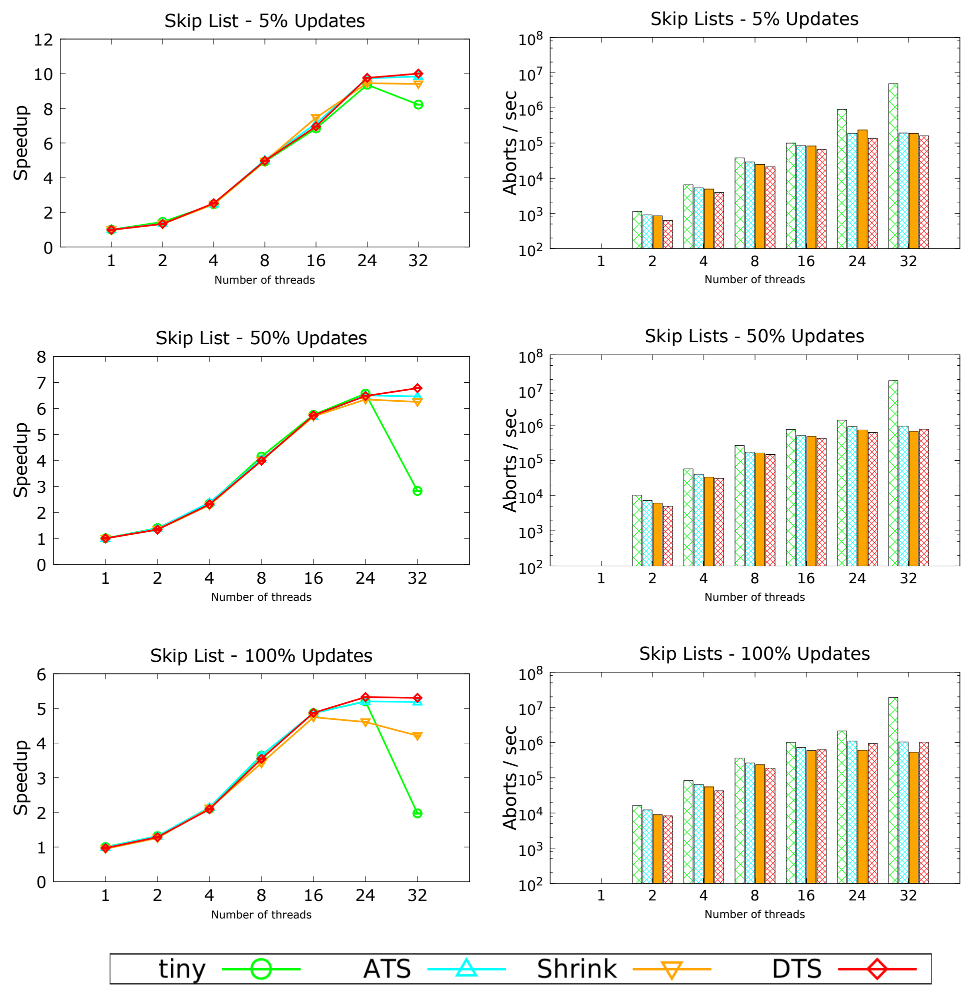
3.3.4 The Effect of the Scheduler on Performance

This section presents a performance comparison between DTS with SRP, TinySTM, ATS and Shrink. The first evaluation uses four micro-benchmarks that are common for the evaluation of STM implementations [20]. These benchmarks randomly insert, delete or search for elements in an integer set implemented as a Skip Lists (SL), sorted Linked Lists (LL), Red-Black trees (RB), or a Hashtable (HS). Insertions and deletions alternate in order to keep the set at an stable size. Each operation is wrapped in a transaction. A read transaction determines whether an element is in the set, and an update transaction, adds or removes elements. Contention is controlled by the size of the set, and the ratio of insertion and deletion operations to read-only searches.

For this experimental evaluation, the sets are initially populated with 2^{12} elements and the throughput, measured in operations per second, is computed after performing random insertions and removals for 30 seconds. Figure 3.7 shows the results of update rates 5%, 50% and 100% for integer sets using Skip Lists. The performance measurements for update rates of 5%, and 50% in integer sets with Red-Black trees, and for update rate of 5% in integer sets of sorted Linked Lists, and Hashtable are shown in Figure 3.8.

Skip List and Red-Black use data structures designed to make it possible to access any element of the set by traversing only a few other elements, and thus may exhibit high potential parallelism. Skip List operations are characterized by a medium number

Figure 3.7: Speedup and amount of aborts per second on integer sets implemented with skip lists – Speedup normalized to the sequential execution



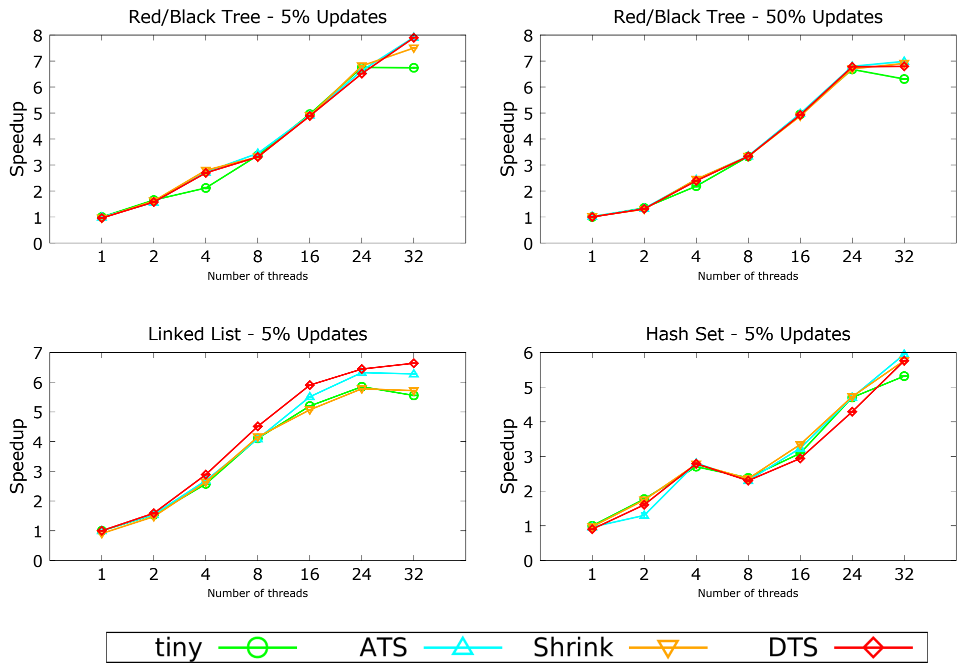
of reads, and a small number of writes. This ratio between reads and writes results in medium-length transactions. In all cases, DTS achieves the best scalability and is followed by ATS. At high contention, Shrink's scalability is affected by its Bloom filter's false positives. This effect is better understood by examining the bar plots on the right side of Figure 3.7 that show the number of aborts per second. The amount of contention increases with the percentage of updates as the difference in the abort ratio between 5% updates and 50% updates illustrates. A curious effect appear at 100% updates: the number of aborts for Shrink for 24 threads or more is slightly lower than for the other techniques — also observed for 32 threads and 50% updates. This reduction in the number of aborts reflects the more aggressive serialization in Shrink which, unfortunately, limits concurrency and does not lead to improved performance as the decreased speedup on the left side of Figure 3.7 indicates.

Red-Black tree operations are characterized by a small number of reads, and a small number of writes. Fewer operations lead to shorter transactions and affects all implementations equally.

Linked-list operations are characterized by a high number of reads, due to the need to traverse the list from the head to the required node, and a few writes. This read-to-write ratio results in long transactions. Moreover, any write to a previously visited element that occurs before a transaction completes can cause a transaction conflict. DTS achieves the best scalability for benchmarks with low-update rates but its performance degrades as the number of threads is increased in benchmarks with high-update rates due to a high number of aborts. This also occurs with other strategies.

Hash Set operations are characterized by a small number of reads, and a medium number of writes leading to short transactions. Moreover, transactions suffer from a high number of aborts due to collisions, linked-list chains, and duplicate inserts that update the memory. The high abort ratio in this benchmark affects all implementations.

Figure 3.8: Speedup on integer sets implemented with Red-Black trees, sorted Linked Lists, and Hashtable – Speedup normalized to the sequential execution



In Figure 3.9 the STAMP benchmark suite is used to compare the performance of DTS with SRP, TinySTM, ATS and Shrink. `Bayes` does not provide reproducible behaviour and displays a large variance. This behaviour has been reported before [11].

`Genome` and `ssca2` have short transactions, very low contention level, and exhibit no change in performance based on the different scheduling policies. Besides, short transactions amplify the overhead of the collection of data for the metrics, and do not favor TM scheduling policies against CM policies.

`Yada` is composed of both, short and long transactions. Out of five static transactions in `yada`, three are extremely short (usually only a single transactional read or write) and do not scale. DTS has little or no impact on applications that have low contention because only a small portion of cycles are wasted in aborts.

The results indicate that DTS reduces the total execution time of benchmarks with high contention (transactional characteristics of each STAMP benchmark were described by Minh *et al.* [36]). `Labyrinth`, for instance, has very long transactions with very large read and write sets. The amount of contention is very high because of the large number of transactional accesses to memory [36]. In addition, due to the long transactional execution delay, there are repeated conflicts with the transactions in other threads. `Labyrinth` is very favorable for heuristics whose main goal is to prevent aborts from happening. In this case, the results achieved by the strategy adopted by ATS is very similar to the results of DTS.

The algorithm in `kmeans` groups objects that are placed in an N-dimensional space into K clusters. The amount of contention among threads depends on the value of K, with smaller values resulting in more frequent conflicts because the probability that two threads are concurrently operating on the same cluster is higher [36]. Therefore, even though `kmeans` has short transactions, the contention level becomes higher when the number of threads increases (above 16 threads), providing many scheduling opportunities that explain the improvement in the performance of DTS.

A comparison between DTS and ATS indicates that when there is a more significant difference in performance, such as in `kmeans` and `intruder`, DTS with SRP outperforms ATS.

3.3.5 Evaluating SRP on High-Contention Workloads

As seen in Section 3.3.4, DTS scheduling with SRP is expected to provide speedup gains when transaction diversity (characterized by their workloads) is reasonably large and contention is high. Higher contention is obtained, for instance, by increasing the number of transactional threads running simultaneously.

In the read-write workloads, there is too much writing for readers to get a benefit from optimistic concurrency, so there is not much scalability to be found with TM. None of the implementations scale well in this experiment (see Figure 3.10). Notice, however, that the average performance of DTS is slightly better than ATS and Shrink, mainly between 4 and 16 threads.

In read-dominated workloads (shown in Figure 3.11), DTS delivers good and robust performance up to 16 threads like TinySTM, and superior performance when compared with ATS and Shrink.

The benefit of the PEW and SRP becomes clear as the number of threads and the sizes of the data sets increase. For configurations *Small* and *Medium* DTS gives similar speedups with TinySTM. For configurations *Big* and *Huge* the length of the average transaction considerably grows because the transaction length increases with the data size. In this scenario, DTS improves over TinySTM, handling with the contention without any performance pathologies and with relatively low overhead.

Figure 3.9: STAMP applications – Speedup normalized to the sequential execution.

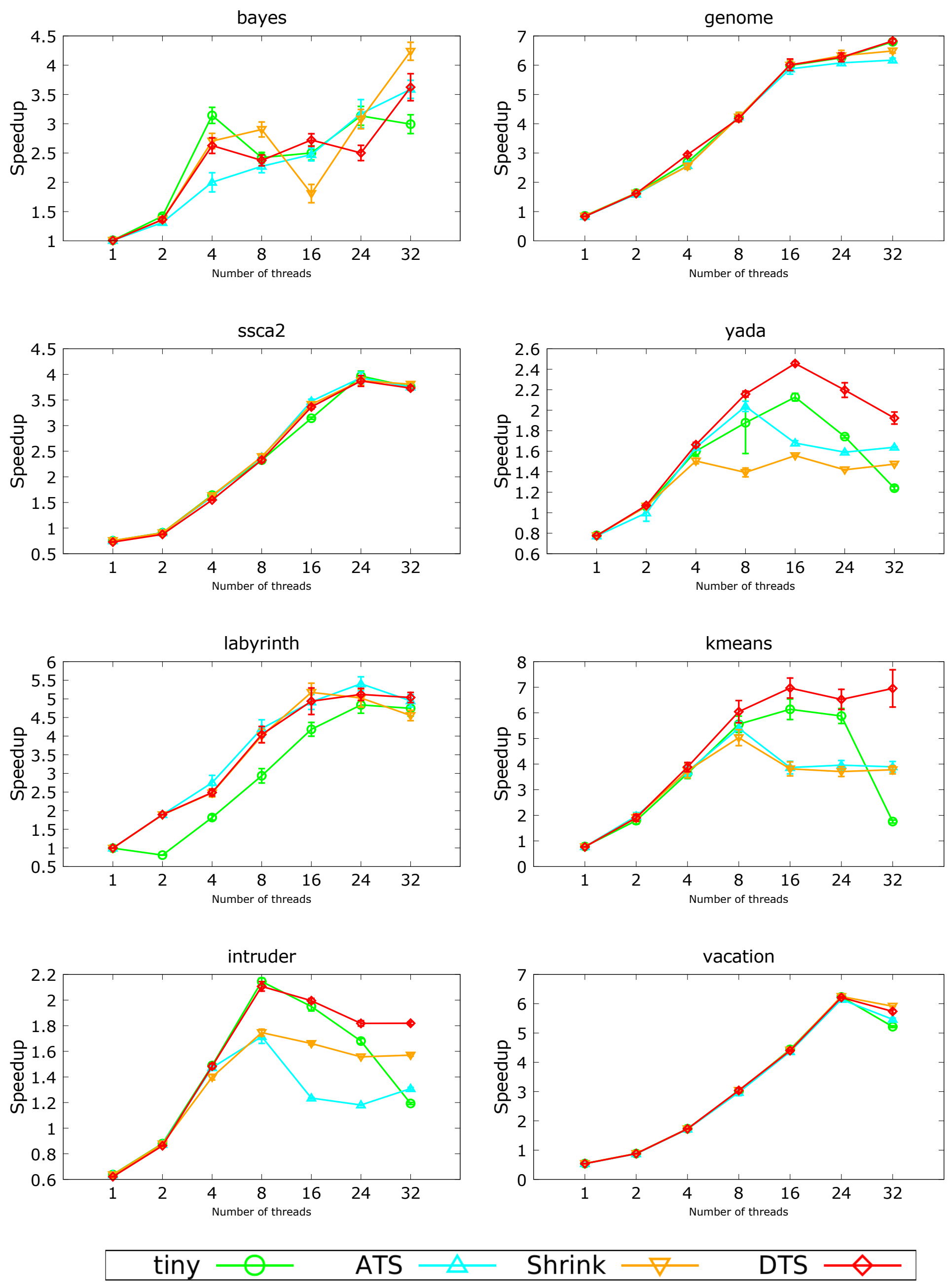


Figure 3.10: Throughput on STMBench7 with read-write workloads.

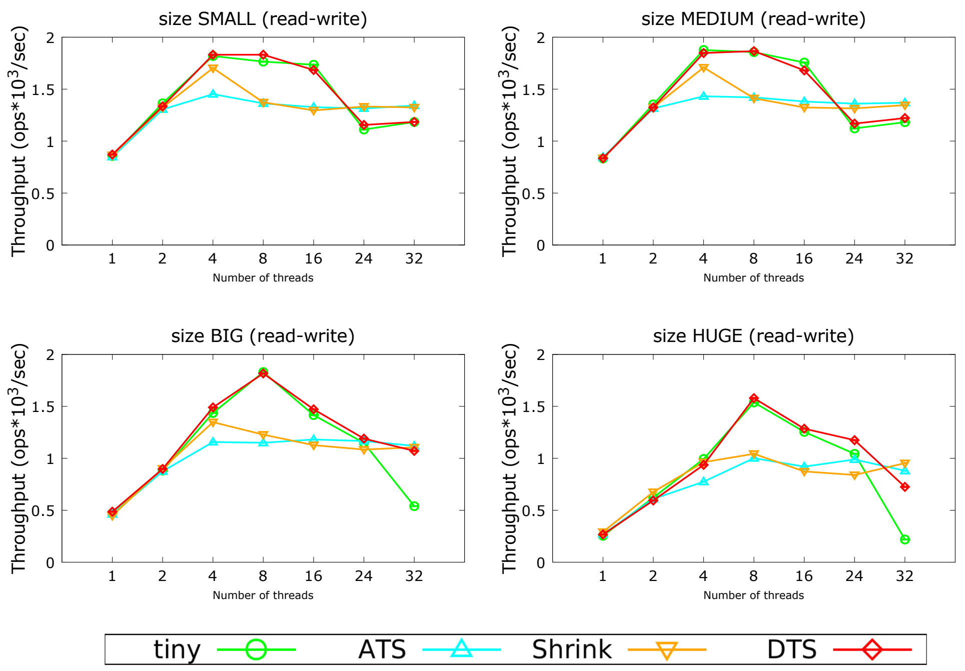
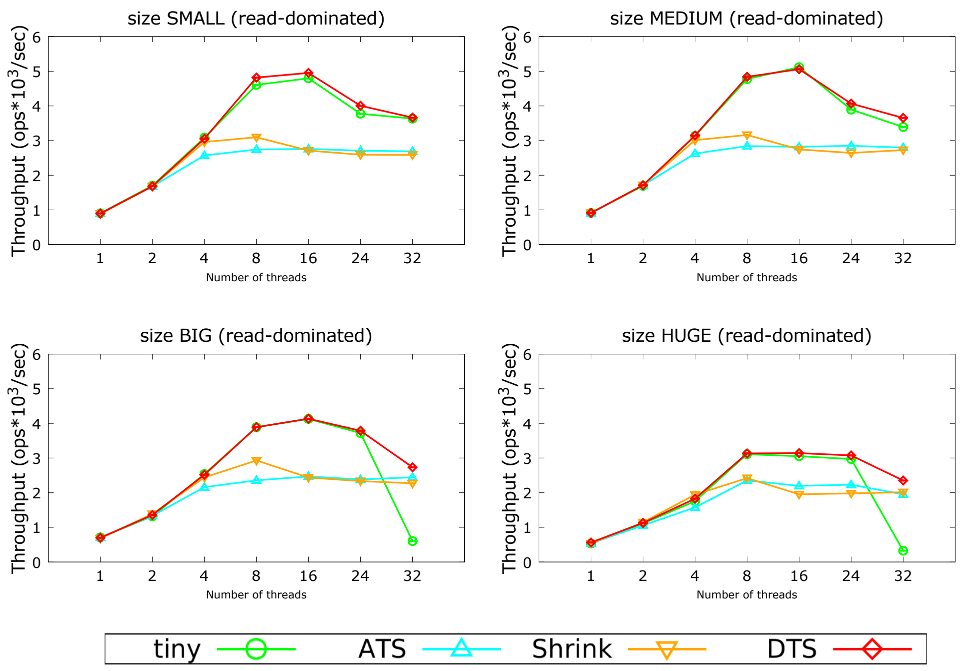


Figure 3.11: Throughput on STMBench7 with read-dominated workloads.



3.4 Related Work

There are three areas of work that are related to the contributions in this thesis: contention managers that decide what to do after a conflict happens, conflict avoidance policies, and transaction schedulers that decide which transaction to execute next.

3.4.1 Contention Management

A host of policies to resolve contention have been proposed during the past ten years [22, 39, 5, 43, 41]. Still, contention managers have only a few choices to deal with transaction conflicts. A contention manager decides which of the conflicting transactions can continue and whether the other transactions involved in the conflict should be aborted or delayed. The contention manager also decides how long a conflicting transaction must wait before it can restart or resume execution. All contention managers are limited to act only *after* a conflict has happened.

To overcome this limitation, scheduling-based contention management approaches have emerged [48, 16, 3, 17, 6, 35]. In special, Attiya and Milani [6] offer a formal description of schedulers that target read-dominated workloads. The new PEW metric is a natural evolution of this research, that provides better information about the history of a transaction execution and allows for better decisions about which transactions should be scheduled next. While this work proposes the use of PEW in DTS to proactively avoid conflicts, PEW could also be used to improve conflict resolutions by contention managers.

Hardware Transactional Memory (HTM) systems are often limited on the information that they can glean from transaction execution and also often rely on a fix hardware-based scheduling policy. Therefore they offer limited scope to implement contention management. For instance, the TM system in the IBM BG/Q machine adopts a policy of performing a set number of retries for an aborted transaction before causing the transaction to serialize, through the acquisition of a global lock, to guarantee forward progress in the system [44].

3.4.2 Conflict Avoidance

To the best of our knowledge, the first scheduling-based conflict avoidance proposal is due to Yoo and Lee [48] (see Section 2.1), named Adaptive Transaction Scheduling — ATS and discussed in Section 3.2.1. Other attempts at scheduling to avoid conflicts include CAR-STM [16], a transactional scheduler that maintains a queue of transactions for each core. When a conflict between two transactions arises, the scheduler inserts the aborted transaction into the aborter's queue. Therefore, the likelihood of repeated aborts is decreased because the aborted transaction can only execute after the execution of the transaction that caused the conflict is complete. CAR-STM proposes proactive collision reduction to exploit the probability of a conflict and to serialize those transactions that are more likely to conflict. However, CAR-STM expects applications to provide a conflict probability (CP). CP is generally difficult to determine because the programs tend to exhibit different execution phases and the CP may vary over time. In contrast, DTS uses the execution history of each transaction to decide which one will execute next and does not require any specific information from applications.

The same approach, with slight variations, is used by Steal-on-Abort [3]. The variation concerns where the aborted transaction is inserted into the aborter's queue (e.g. at the tail or at the head).

D. Choi *et al.* [10] propose a method called Conflict Avoidance Scheduling — CAS that aims at preventing conflicts in high-contention scenarios. In CAS, threads that execute transactions with high probability of conflicts are grouped together. Based on the group information, concurrent execution of threads, in the same group, is restricted by means of serialization.

Maldonado *et al.* [35] investigate kernel-level support for serializing contention management, guided by the motivation that user-level implementations are costly as they require system calls. An inconvenience of this approach is that the OS kernel needs to be modified.

Recent works have focused on providing more accurate prediction and resorting to serialization only as the last option. For instance, Atoofian [4] propose a Speculative Contention Avoidance — SCA to prevent conflicts in TMs. SCA dynamically controls the number of concurrently executing transactions and serializes those transactions that are likely to conflict. This technique is built upon the concept of contention locality: the likelihood that a previously aborted transaction will fail again in the near future. The premise is that contentions are highly predictable in TMs. SCA exploits history-based predictors to speculate the chance of conflicts in the future. SCA dynamically adjusts the number of executing transactions based on feedback received from the predictors. If there is high confidence that a transaction will conflict then SCA serializes the transaction, reducing the number of concurrent executing transactions. However, performance improvement is possible only if contentions are predicted accurately by SCA. In the event of a mis-prediction, SCA serializes transactions unnecessarily resulting in program slowdown. The new PEW metric could offer a more accurate prediction for a system such as SCA.

3.4.3 Transaction Schedulers

Blake *et al.* [7] propose a prediction mechanism based on the past conflict history of transactions. Their Proactive Transaction Scheduling — PTS system keeps a conflict table that stores the likelihood of a conflict between any pair of transactions in the system. The scheduler relies on this information to decide whether a transaction should proceed or should be serialized. In PTS a new thread is executed on behalf of an aborted thread when conflicts occur. However, if the number of conflicts between threads is large, the thread swapped in place of the aborted thread is also conflicted. On the contrary, DTS does not perform the swapping operation on aborted threads, but aims at finding the transaction with the best execution priority of all available transactions.

Excessive serialization of memory transactions may limit concurrency too much and hurt performance. Herber *et al.* [27] investigate how serializing CM influences the performance of STM systems. Specifically, they have studied serialization's influence on STM throughput and efficiency as the workload's level of contention changes. The characteristics of TM workloads generated by real applications may vary over time. To achieve good performance, CM algorithms need to monitor these characteristics and change their behaviour accordingly.

The approach provided by DTS is in sharp contrast with the majority of the related work becasuse DTS aims to find the best execution priority of the available transactions instead of serializing then.

Chapter 4

Serialization Techniques for HTM

On of the goals of this research was to present a performance evaluation of the Intel's TSX from the application perspective, providing a precise evaluation of the strengths and weaknesses of this architectural feature[1] followed by well-design serialization techniques to provide forward progress. These techniques, or policies, are implemented by Serialization Managers, which plays a similar role to the Contention Manager in an STM, guaranteeing forward progress and attempting to improve application performance. These techniques are described in Section 4.1.

The main finding of this performance study is that Intel's TSX performance is most sensitive to i) the *transaction footprint*, defined as the number of shared write accesses executed inside a transaction; ii) the *working-set size*, defined as the number of distinct memory locations accessed — read from or written to — inside a transaction;[2] iii) the *transactional write ratio* (or *pollution*), defined as the ratio between the number of shared writes and the total number of shared accesses in a transaction; and iv) the *contention level* in a transactional application. The contention level is the probability that dynamic transactions will abort due to a conflicting access.

This sensitivity is first demonstrated through an analysis that isolates the effect of each TM application characteristic on performance using Eigenbench [30] and CLOMP-TM [40] to identify the constraints imposed by TSX. Next, the study evaluates the performance of TSX using the more realistic STAMP benchmark suite [36].

Based on the experimental results, the following application features are likely to yield performance gains when using Intel's TSX:

[1]Preliminary performance evaluation for IBM Power8 can be seen on Appendix A

[2]A more precise metric is the number of distinct cache lines that are occupied by the working set of a transaction.

- In a low-contention scenario, if the *transactional footprint* is small, then it is preferable to convert multiple transactions into a single transactional region. This conversion amortizes the overhead necessary to execute a transaction as long as the *transaction footprint* do not exhaust the capacity of cache lines. In high-contention scenarios, it is preferable to maintain transactions with smaller footprint to reduce wasted work.

- *Transaction footprint* and *contention* are the most important characteristics that dictate TM performance. These characteristics are strongly influenced by *temporal locality* and *pollution*. To obtain performance gains, a designer should create data structures that increase temporal locality and reduce pollution. For example, data structures based on hash often have a better performance than linked lists because they tend to reduce pollution.

- The performance of Intel's TSX is also sensitive to the policy applied on the fallback path to ensure forward progress. The right choice of fallback policy can avoid unnecessary serialization thereby allowing more concurrency and improving performance.

4.1 Forward-Progress Policies

Due to inherent architectural limitations, Intel's TSX is known as 'best-effort' HTM, i.e., it does not guarantee that a transactional execution will eventually commit. One possible reason for a transaction never to succeed is because its data footprint does not fit in the relatively small L1 data cache. The expectation is that a software fallback handler is provided by the policies that guide the TM application, mimicking an unbounded TM system, similar to an STM, while providing forward progress guarantees for transactions that fail. Generally, the strategy adopted is to retry the execution of the transaction, with or without a time delay, to attempt to complete the transaction execution speculatively. A time delay, often called *backing-off*, can avoid the pathological pattern called *Convoy* [3] [8]. In the face of persisting failure, a transaction must be completed by running it in a non-speculative execution mode. A common solution is to acquire a global lock to prevent other transactions from committing concurrently. The switching to non-speculative execution is controlled by the Serialization Manager, guaranteeing forward progress and attempting

[3]A convoy phenomenon occurs when multiple transactions contend repeatedly for the same resource. Each time a transactional thread attempts to execute and fails, it forces a context switch. The overhead of repeated context switches degrade overall performance.

to improve application performance. The experimental prototype in this work implements three forward-progress policies, namely *MaxRetry*, *Backoff* and *SerControl*.

4.1.1 MaxRetry Policy

MaxRetry is the simplest way to ensure forward-progress. It simply limits the number of times that a dynamic transaction can be retried to a predefined threshold N. A transaction that exceeds its retry budget must be executed while holding of a global lock. An empirical evaluation for values of N produced the best results, for the benchmarks studied, N equal twenty. *MaxRetry* is not a policy used in any actual HTM system. It is included in this study to enable an evaluation of the effect of backing-off in reducing the Convoy pathology described by Bobba *et al.* [8].

4.1.2 Backoff Policy

Backoff is a forward-progress policy similar to MaxRetry, except that the aborted transaction waits for a time delay before restarting. The delay duration is chosen uniformly at random from a range whose size increases exponentially with every restart. This is continuously done up to the limit of (predefined) N consecutive aborts. Experiments were conducted for different values of N and the best results were obtained for N equal to twenty. After that, the transaction must be executed under a lock protection.

4.1.3 SerControl Policy

SerControl is a new forward-progress policy that selects one of three actions upon a transaction abort: *retry*, *backoff* or *serialize*. The *SerControl* policy examines the return code provided by the hardware in the EAX register, which contains various status bits indicating the cause of the abort. If the transaction aborted because of a conflict with other transactions or because of capacity limitations, the action selected is *backoff*. This strategy makes sense because a capacity abort may be caused by competing transactions vying for the same storage resources. To prevent frequent aborts due to capacity overflow *SerControl* serializes a transaction that has suffered two consecutive capacity aborts. [4] If the cause of an abort is other than conflict or capacity, the action is *retry* for three consecutive aborts [5] before changing to *backoff*. The idea is that aborts, such as an abort

[4]This safeguard is needed because an abort reported as a capacity abort may be due to temporary competition for resources.

[5]This threshold was determined through a series of experiments, with the STAMP benchmarks, that examined performance, serialization rate, and number of aborts due to different causes.

caused by a page-fault, may not occur again if the transaction is retried immediately. The *SerControl* policy limits retries to a threshold of N consecutive aborts. The value of N used in the evaluation was twenty for a fair comparison with the other two policies. After N retries, a transaction must be executed under the control of a lock — a *serialize* action.

Intel's TSX exhibits lower overhead when compared with STM. The prototype keeps this overhead low by using the minimum data structure required to manage a transaction. This data monitors the number of retries and the number of capacity-induced aborts, and selects the *SerControl* policy action and the time delay (see the pseudo-code in Listing 4.1). Henceforth, TM overhead includes the overhead imposed on the transaction management by the forward-progress policies.

Listing 4.1: Low-Overhead High-Accuracy SerControl Policy

```
upon TM_START
  int backoff = MIN_BACKOFF;
  int (try = 0, status = 0);
  int (conflict = 0, capacity = 0);
  long wait = 0;

upon TM_BEGIN
  if (++try>=MAX_RETRY) status = SERIALIZE;
  else
    case (status = _xbegin())
      XBEGIN_STARTED: ;
      XABORT_CONFLICT:
        capacity = 0;
        status = (++conflict>=MAX_CONFLICT) ? BACKOFF : RETRY;
      XABORT_CAPACITY:
        status = (++capacity>=OVERFLOW_CAPACITY) ? SERIALIZE : RETRY;
      others:
        capacity = 0;
        status = (try>=MAX_CONTROL) ? BACKOFF : RETRY;
    esac
  fi
  if (status==SERIALIZE)
    status = get_spin_lock();
  fi
  if (status==IN_SPIN_LOCK) or
     (status==XBEGIN_STARTED and spin_lock==FALSE)
    begin_transaction();
  fi

upon TM_END
  if (status==IN_SPIN_LOCK)
    release_spin_lock();
    else
      commit_transaction();
    fi

upon TM_ABORT
    if (status==BACKOFF)
      wait = get_new_time_delay();
      while (wait--) ;
      backoff++;
    fi
```

4.2 Experimental Evaluation

This section presents an experimental evaluation using a prototype implementation of the *SerControl*, *MaxRetry* and *Backoff* forward-progress policies on top of Intel's TSX processor to guide TM applications. The main findings about the policies are:

- Simple policies, such as *MaxRetry* and *Backoff*, have the potential to deliver performance in RTM because of their low overhead to monitor transaction execution and of their simple approach to decide when to retry an aborted transaction. However, the experimental results indicate that these policies are also sensitive to the tuning of the parameters used to decide when to restart or serialize the execution of a transaction.
- Among the policies evaluated, *SerControl* is the most successful in delivering performance for transactional applications for RTM due to its strategy of reducing potential conflicts and, consequently, the number of aborts. Moreover, strict adherence to Bobba's suggestion of exponential backoff [8] can be detrimental to some applications.

4.2.1 Experimental Infrastructure and Methodology

This experimental evaluation uses a computer with an Intel Xeon Processor E3-1200 v3 (4 cores with 2 hyper-threads per core, 8 threads in total) clocked at 3.10 GHz with 8 GB of RAM. Each core has a 32 KB L1 Data Cache and a 256 KB L2 Cache. The operating system is Ubuntu Server 13.10 amd64.

The results presented in all experiments are averaged over twenty executions and the graphs show a 95% confidence interval. In the graphs that show an upper-bound speedup, this bound represents the case of a multi-threaded execution without the protection of a TM system. This measurement serves as a hypothetical upper limit for the available performance and is useful to show the overhead of the TM system when independent transactions are running (zero contention). In, the experiments that estimate the upper-bound speedup, the atomic execution of critical sections is not guaranteed and the results could be incorrect. Therefore, it is said that the system is executing in unprotected mode. Unless noted, all speedups are normalized to the execution time of the corresponding best sequential version.

4.2.2 Eigenbench Results

Experiments with the Eigenbench [30] micro-benchmark enable a characterization of Intel's TSX RTM and provide significant insight on its strengths and weaknesses. The Eigenbench is designed to enable independent exploration of the properties [6] of a TM application shown in Table 4.1. Each instance of the Eigenbench benchmark can be thought of as a point in a multi-dimensional space defined by these properties.

Table 4.1: TM properties

Property	Definition	Empirical values
Transaction length	Number of shared accesses per TX	30 words
Working-set size	Size of frequently used memory	32 KB/thread
Pollution	Fraction of shared writes to shared accesses	10%
Temporal locality	Probability of repeated address per shared access	0 %
Contention	Probability of conflict of a transaction	0 %
Predominance	Fraction of shared access cycles inside TX (not explored)	100%
Density	Fraction of non-shared cycles outside TX (not explored)	100%

The methodology to use Eigenbench to discover characteristics of the Intel's TSX RTM implementation consisted of an initial empirical exploration of the space defined by the values of the properties listed in Table 4.1. This exploration led to the discovery of a "baseline configuration" where the values listed on the rightmost column of Table 4.1 were used. With these values fixed, a more systematic exploration of the space varied the value of a single property at a time while maintaining the other values at the baseline configuration. In occasions where this exploration indicated that there was potential for better performance when two or more properties were moved away from this baseline, those combinations were also tried.

The graphs in Figure 4.1 display the trends for each of these experiments using the three policies: *MaxRetry*, *Backoff* and *SerControl*. The horizontal axis denotes the value

[6]For the Eigenbench, transaction footprint is a derived property, calculated from *transaction length* × *pollution*.

of the property that is being controlled and the vertical axis denotes the speedup results for four threads, normalized in relation to the sequential execution of the same program. Except for the experiment reported in the graph in Figure 4.1(g), independent transactions are used to evaluate the overhead of the TM system. To show the overhead, a curve for a performance upper bound is also included. This upper bound is estimated by running the sequential version in parallel "unprotected" with no transactional overhead from fallback-handling code.

An important limitation of HTM implementations is the amount of speculative state that the HTM system can store. RTM can only successfully complete speculatively transactions that have a small footprint because all speculative state is stored in the relatively small L1 data cache. The experiment reported in Figure 4.1(a) demonstrates this limitation. The *SerControl* policy performs better than *MaxRetry* and *Backoff* for a pollution of 1% thanks to its strategy of reducing potential conflicts and hence imposing a lower demand on the L1 data cache in comparison with the other policies. However, the *SerControl* performance degrades quickly for transaction lengths greater than 60 words[7] for a pollution of 10%. The lower performance for smaller transactions (left side of the plots) indicates that the TM overhead becomes significant. A separate measurement, executing on a single thread, confirms this observation. For an empty transaction, or a transaction length equal zero, the transactional execution time is twelve times higher than the sequential execution that has no overhead. For transaction length equal to 5 words this time drops to 2 times of the sequential execution. When transaction length reaches 30 words, this overhead is almost totally amortized. The graph on the left in Figure 4.1(b) shows that *SerControl* perform better than *Backoff* between 60 and 140 words while the graph on the right in same figure points out that *SerControl* is dramatically more reluctant to serialize when capacity aborts occur, confirming one of the payoffs of the *SerControl* policy.

The graphs in Figure 4.1(c) show the effect of different *working-set* sizes for transaction lengths equals to 30 (left plot) and 50 (right plot) words. *MaxRetry* is the policy that suffers most due to the high number of aborts and retries and hence the overhead is more pronounced in short transactions. There is a dramatic speedup drop starting at 256 KB/thread — even for the upper-bound curve — because the data processed by the transactions exceed the capacity of the L2 cache. For TX length equal thirty the most interesting comparisons are the curves for *MaxRetry* and *Backoff* on the left of the plot and the curves for *Backoff* and *SerControl* on the right of the plot. To better understand these performance trends, Figure 4.1(d) reports the ratio between the number of transactions that abort and the number of transactions that commit on the left, and the ratio between

[7]One (1) word equals four (4) bytes in the target machine.

the number of transactions that serialize and the number of transactions that commit on the right. For smaller working sets, always backing-off is a good policy because it eliminates the convoy-effect aborts resulting from *MaxRetry*, as the abort ratios on the left plot of Figure 4.1(d) indicate. As the working-set size becomes larger, the selective back off used by *SerControl* is a better policy. However, with longer transactions and larger working-sets, selectively backing-off does not prevent the increasing number of aborts and serializations as shown by the plots on the right of Figures 4.1(c) and 4.1(d).

A study of the effect of temporal locality on performance is shown in Figure 4.1(e). The probability of address repetition varies between 0 (no address is used twice in a transaction) and 1 (a single address is used inside each transaction). The initial drop in performance for both *Backoff* and *SerControl* for a transaction length equal 50 occurs as addresses start being repeated. This may be explained either by the associative effect of the L1 data cache — the Eigenbench generates random addresses that might cause more potentially useful data elimination from the cache.

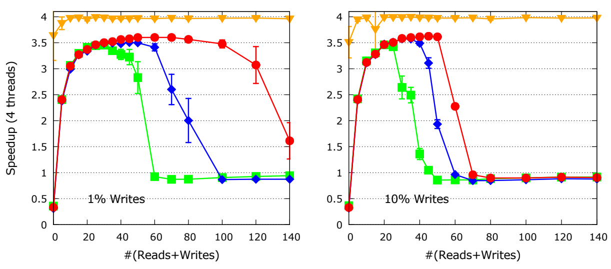
Until the *transaction footprint* exhausts cache lines, at about 32% of shared writes for a transaction length of 50 (i.e., 64 bytes), the *SerControl* strategy of avoiding potentially unproductive retries yields higher speedups than either *MaxRetry* or *Backoff* (see Figure 4.1(f)). The drop in speedup for *SerControl* for larger transactions is due to the limited capacity in the cache to store speculative state.

Although critical details are not available, there is sufficient information to speculate about the nature of Haswell's TM. Haswell's TM uses the L1 data cache to track the write-set. While it stores transactional reads in the L1 data cache, it also uses a separate mechanism, perhaps *Bloom Filters* [9], to track speculative reads that have been evicted from the L1 data cache. The caches and fill buffers are competitively shared by any active threads. However, store operations only need to write the address (and eventually the data) into the store buffer while load operations must write into the load buffer and also probe the store buffer to check for any forwarding or conflicts. It appears that the shape curve of *MaxRetry* policy on the left graph in Figure 4.1(f) is due to the difference in the conflict detection mechanism between read- and write-set and is stimulated by increased competition due to the Convoy effect.

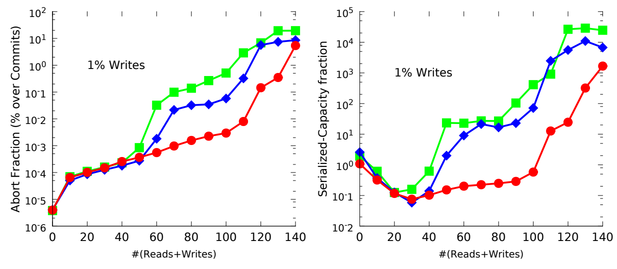
Finally, how does contention affect performance in the Intel's TSX? Experiments with transaction lengths equal to 20, 30 and 40 words — varying the fraction of writes per transaction between 5% and 100% — help address this question. The x-axis shows the value of expected contention. The graphs in Figure 4.1(g) indicate that the TM performance drops quickly with long or dirty transactions. All three policies are very sensitive to contention greater than 4%, although the *SerControl* policy performs better if the transaction length fit between 30 and 40 words. This level of contention was achieved for

Figure 4.1: Analysis of RTM using Eigenbench

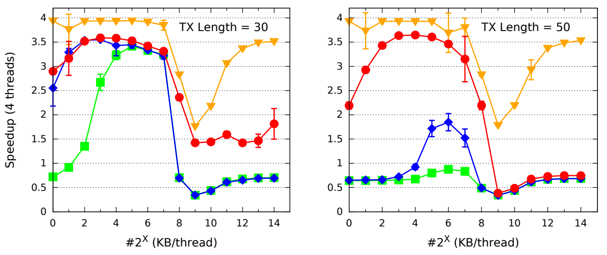
(a) Transaction Length



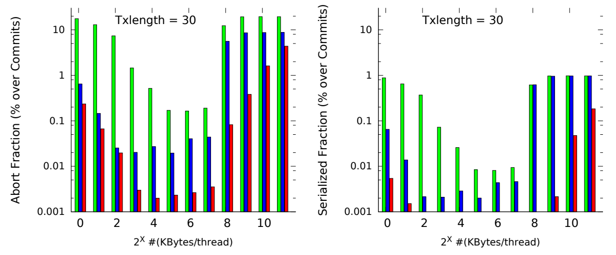
(b) Analysis of outcomes for Transaction Length



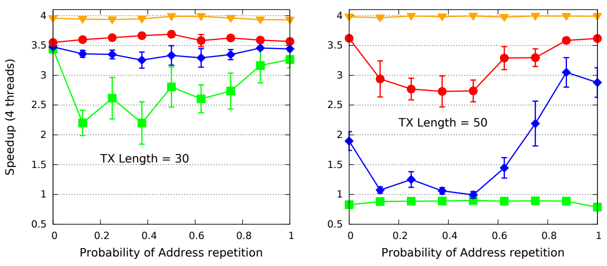
(c) Working-set size



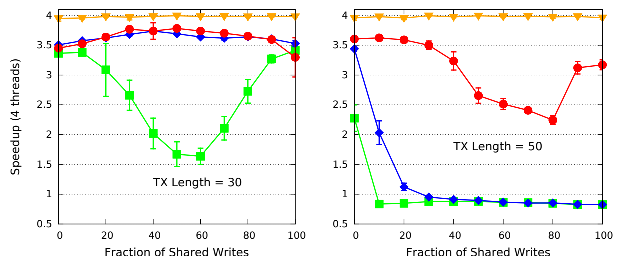
(d) Analysis of outcomes for Working-set



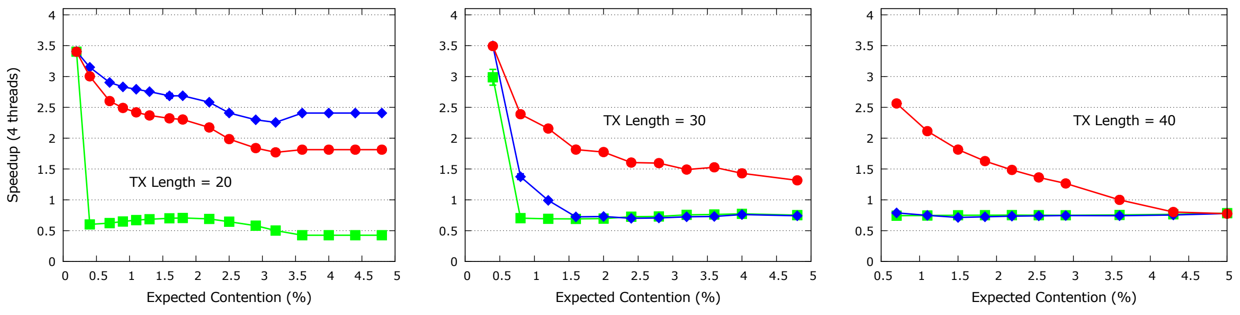
(e) Temporal Locality

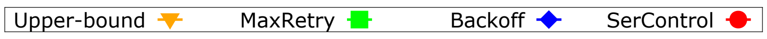


(f) Pollution



(g) Contention





write-ratio of 30% to 40%, which implies in a transaction footprint between 48 and 64 bytes. This reveals a severe resource constraint for TSX, limiting the footprint to the size of the L1 data cache.

4.2.3 CLOMP-TM Results

Section 4.2.2 showed that, to make an effective use of RTM on Intel's TSX, we need to define the design space that can be exploited by the TM applications. For this, it is necessary to complement this study with a more appropriate analysis of the performance behaviour of RTM at different TM footprints. The CLOMP-TM benchmark [40, 47] is used for this analysis.

CLOMP-TM is a synthetic memory-access generator that emulates the synchronization characteristics of high-performance computing applications. It is specifically designed to expose the range of properties needed to characterize scientific workloads. It was created to mimic the application characteristics of several large scale, multi-physics applications used in production at the Department of Energy laboratories in the USA.

CLOMP-TM uses constructs such as atomics or OpenMP-based constructs (*omp critical* and *omp atomic*) to synchronize OpenMP threads with the same level of abstraction. We adapted CLOMP-TM to use Intel's TSX RTM with the prototype implementation of the *SerControl* policy to evaluate its performance on small and large footprints, and also to compare the performance obtained with the fine-grained lock implementation in both low- and high-contention scenarios.

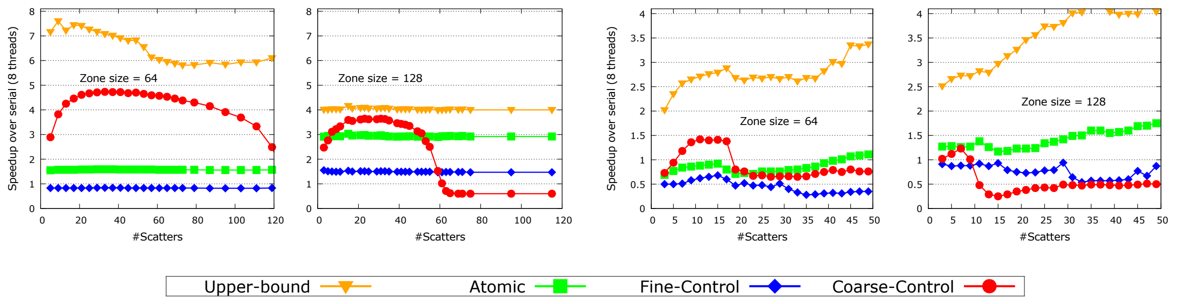
CLOMP-TM resembles an unstructured mesh with a set of partitions. Each partition holds a linked list of zones. To vary the pressure on the memory system, the size of these zones were configured with two values: 64 bytes, to fit the size of the cache line, and 128 bytes. Also, two different contention levels were used: low contention and high contention. Contention occurs when multiple threads update the same zones (for the contention scenarios, the memory access patterns used were *Adjacent* and *FirstParts*, respectively [8]). Each zone is pre-wired to deposit a value to a set of other zones, called *scatter zones*, which involves reading the coordinate of a scatter zone, doing some computation, and depositing the new value back to the scatter zone. These value deposits were synchronized in two ways:

- `Fine-Control` is a small-footprint transaction with a single scatter zone value update. `Fine-Control` resembles the case where a lock-prefixed instruction is used to enforce atomicity on a single variable.

[8]For more information on memory access patterns [40].

Figure 4.2: Analysis of RTM using CLOMP-TM

(a) Low Contention (b) High Contention



- `Coarse-Control` is a large-footprint transaction where all scatter zones are updated for each zone. `Large-TM` resembles the update of multiple variables in one critical section.

Figure 4.2 show the results for 8 threads. Atomic (fine-grained lock) denotes the use of the OpenMP construction **#pragma omp atomic**. The X-axis denotes the number of scatters for each zone, and at each scatter count, the speedup is measured over the execution time of the corresponding serial version. For clarity, the graphs show only the results for the *SerControl* policy for small- and large-footprint transactions (`Fine-Control` and `Coarse-Control`, respectively).

The results in the graph in Figure 4.2(a) lead to the following best-practice guideline: in a low-contention scenario it is preferable to convert multiple lock acquisitions or critical sections into a single transactional region, to better amortize the TM overhead, because the transaction footprint does not exhaust the available cache-line capacity. In high-contention scenarios, it is preferable to maintain transactions with smaller footprint, as evidenced by the graph in Figure 4.2(b).

Two additional best-practice guidelines arising from these results are: (i) *transaction footprint* and *contention* are the most important characteristics that dictate the TM performance on Intel's TSX as evidenced by the graphs on Figures 4.1(a) and 4.1(g). As shown in the graphs of Figures 4.1(e) and 4.1(f), one can reduce the transaction footprint by reducing the pollution in transaction or by increasing the temporal locality. These effects can be achieved by carefully designing the data structures of the application. (ii) Library-level support for TM matters. Moreover, try to use a serialization policy similar to the *SerControl* policy to avoid unnecessary serialization and to improve performance.

4.2.4 How does TSX perform with a more realistic benchmark?

This section uses the well-known, and widely used, STAMP [36] benchmark suite to evaluate the behaviour of real applications across the input-size dimension, similar to how Eigenbench and CLOMP-TM were evaluated previously. This study starts with the STAMP recommended configurations and data sets for use in real machines (for detailed information, see Table IV on [36]) and varies the input size to understand if real applications see the same effects seen in the synthetic benchmarks regarding working-set and transaction size. This is an unconventional use of the STAMP benchmarks, but it is very appropriate for this evaluation to understand the constraints that Intel's TSX creates for applications and application performance.

Figure 4.3 shows the speedup results of the *MaxRetry*, *Backoff* and *SerControl* policies, running on four threads, over the sequential execution in the same configuration.

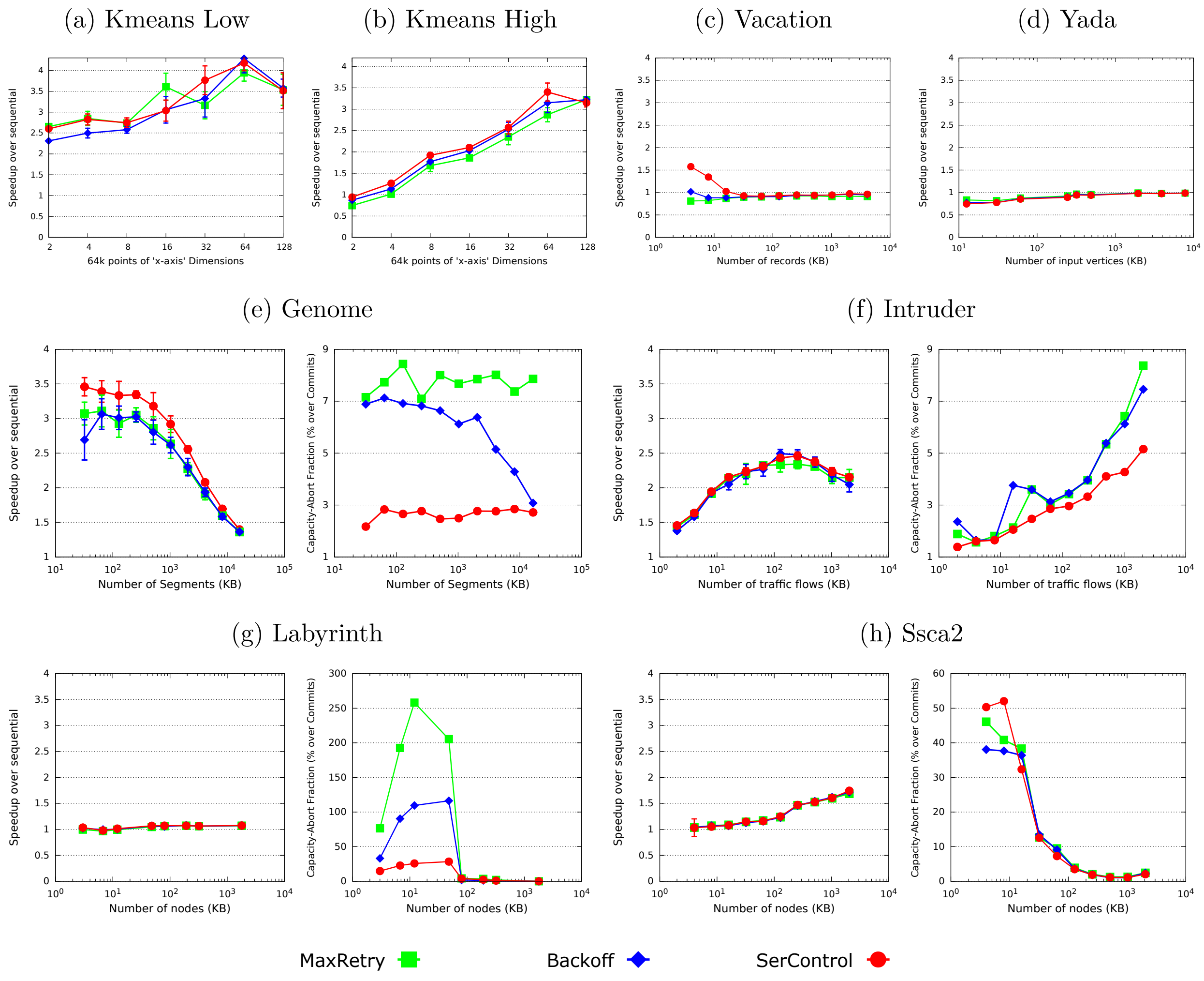
For `Kmeans` the size of the transaction is proportional to the dimensionality of the space. Thus, we fixed the number of points (64 Kbytes), and vary the number of dimensions. `Kmeans` in low-contention scenario (the graph in Figure 4.3(a)) showed good performance results with the recommended parameters. `Kmeans` also exhibit high temporal locality and this further emphasizes the performance behaviour. However, it falls with increasing contention, as shown in the high-contention scenario (the graph in Figure 4.3(b)).

For `vacation`, we vary the number of records and, therefore the working-set size. `Vacation` (the graph in Figure 4.3(c)) showed a poor performance due to large-footprint transaction, the governing characteristic.

The `yada` (Yet Another Delaunay Application) benchmark implements Ruppert's algorithm for Delaunay mesh refinement. In the STAMP benchmark suite, `yada` does not come with enough datasets to test the behaviour of the application with different input sizes. We built a new dataset using the triangle application [42] through successive refinements from the data source `ladder` and we obtained data ranging from 32KB to 4MB on the number of input vertexes. `Yada` (the graph in Figure 4.3(d)) has relatively long transactions and a moderate amount of contention. With the increase of input vertexes, the transactions lead to large read- and write-sets and do not scale.

For `Genome` (the graphs in Figure 4.3(e)), we vary the number of gene segments. Even though the transactions in genome are of moderate length and have moderate read- and write-set sizes, the performance for `Genome` is very sensitive to increasing the number of segments — there is a dramatic performance drop because of longer transactions and larger working-set. The fraction of capacity-aborts per commit is lower for *SerControl* leading to slightly better performance. However, *SerControl*'s strategy is not effective to

Figure 4.3: STAMP applications – Analysis on Speedup and fraction of aborts to different input sizes.



reduce the convoy effect for `genome`.

For `Intruder` (the graphs in Figure 4.3(f)), we vary the number of traffic flows. The limited performance comes from the large-footprint transaction and moderate-to-high levels of contention. The change in traffic flows modifies the working-set size, increasing the fraction of capacity aborts. The selective backing-off strategy of *SerControl* reduces the fraction of capacity aborts, as shown at the graph on the right, but is not sufficient to improve the performance when compared to *Backoff* .

For `labyrinth` (the graphs in Figure 4.3(g)), we vary the number of dimensions that change both working-set size and transaction length. `Labyrinth` has very long transactions with very large read- and write-sets. The amount of contention is very high because of both the large number of transactional accesses to memory and the number of capacity-induced aborts, even for the *Backoff* policy. There is no performance gain in `labyrinth` for TSX with any of the policies.

For `ssca2`, we vary the number of nodes in the graph. Short transactions govern the performance of `ssca2` (the left graph in Figure 4.3(h)). The increase in working-set size has little or no impact on application performance because only a small portion of time is spent in transactions. Also, differences in capacity-induced aborts over commits among the policies only appear for small working-set size (the right graph in Figure 4.3(h)).

The results in this section highlight the challenges faced by Intel's TSX on the STAMP benchmark suite. An analysis reveals that 70% to 80% of the aborts in STAMP are due to the architecture, not the application: (*e.g.*, page faults, system calls). In programs with short run times, these kinds of aborts appear to be have a dominant effect in the experiments, limiting the scalability of applications.

4.3 Related Work

Yoo et al. presents an evaluation of Intel's TSX for High-Performance Computing [47] . They describe that the first implementation of HTM in Haswell processors has significant performance potential. Through their work with the benchmarks and applications, they also investigate some preliminary techniques to best utilize Intel's TSX, in particular, lockset elision and transactional coarsening, but they do not provide library-level support for TM. [9]

Goel, B. et al. presents a detailed evaluation of RTM performance and energy expenditure [21]. They compare RTM behaviour to that of the TinySTM software transactional memory system, first by running micro benchmarks, and then by running the STAMP benchmark suite. They conclude that the system which performs better depends heavily on the workload characteristics. They also conducted a case study of two STAMP applications to assess the impact of programming style on RTM performance and investigate what kinds of software optimizations can help overcome RTM's hardware limitations.

Wang et al. presents an Intel's TSX performance characterization using a simple array access micro-benchmark, identifying several important trends, such as, the relationships between, transaction size, write ratio inside transactions, retry count, and transaction abort rate and performance [45]. Our own study complements their work, by going into

[9]We are integrating the proposed *SerControl* policy within `libitm`, the TM library of the GCC compiler. This will allow achieving transparency to developers while preserving ease of use of TM.

more depth on the limits of TSX, and introducing the *SerControl* policy that outperforms a simple MaxRetry or even the backing-off style policy.

Diegues et al. conducted a study on TM comparing different locking techniques, hardware and software TMs, as well as different combinations of these mechanisms, from the dual perspective of performance and power consumption [15]. Similar to our results, but without exploring forward progress policies, they shed a mix of light and shadows on currently available commodity HTM, identifying workloads in which HTM clearly outperforms any alternative synchronization mechanism, and showed that current HTM implementations suffer of restrictions that narrow the scope in which these can be more effective than state of the art software solutions.

Also, Diegues and Roman showed an approach to self-tuning the number of attempts to reschedule the transaction to Intel's TSX motivated by the fact that no single configuration of the software fallback can perform efficiently in every workload and application [14]. This exploration is orthogonal to the study presented here, showing how online-learning can be used to adjust at runtime the number of rollbacks before serialization granted to each transaction based on lightweight profiling. Their policy, TUNER, uses Upper Confidence Bounds — UCB to select a strategy for adjusting the number of retries given to a transaction when a capacity abort occurs. The goal is to try to distinguish between transient capacity failures caused by momentous cache addressing and persistent capacity failures caused by excessive amounts of required speculative state. The double-check in *SerControl* is a simple heuristic that attempts to achieve the same effect. The authors also used an exploration technique similar to hill climbing/gradient descent search to tuning the number of attempts, a problem not explored in this work.

As we have seen, most applications of the STAMP benchmark suite performs poorly in HTM due to the constraints of hardware (see section 4.2.4). Rei Odaira at IBM Research, in Japan, conducted experiments with a couple of optimizations that they found beneficial on HTM [32]. For instance, in intruder, to reduce transaction footprints, concurrent hash maps and red-black trees are used instead of red-black trees and linked lists, respectively, without changing any semantics. In kmeans, to avoid false sharing, each cluster occupies dedicated cache lines. In vacation, concurrent hash maps are used instead of red-black trees, as in intruder. Except for bayes, they have been used the thread-local memory allocator attached with the original STAMP benchmarks. The thread-local memory allocator avoids contention in malloc(), but it does not support free(). We have not had the opportunity to evaluate the proposed optimizations, in addition to runs off a little in our goal.

Chapter 5

Multi-dimensional Evaluation with htm-*pBuilder*

The previous chapter (Sections 4.2, 4.2.2, 4.2.3, and 4.2.4) presented results from an extensive experimental evaluation based on Eigenbench, CLOMP-TM, and the STAMP benchmarks. In Section 4.2.2 transaction properties are varied but the parameters for each serialization policy, such as the maximum number of retries, are constant. Section 4.2.4 uses the STAMP benchmark for a similar evaluation with fixed policy parameters.

This chapter presents a new tool, called htm-*pBuilder*. It was initially designed to act as a wrapper over Eigenbench and allow independent exploration of the parameters of a given fall-back policy. We used this new tool to explore a range of values for the parameters of the *SerControl* policy. Appendix C presents the configuration parameters used in the experiments reported in this chapter. However, other interesting features, also presented in Appendix C, gives this tool more expressive power than found in its predecessor, Eigenbench, and thereby it allows developers to produce a faster and more accurate assessment of the behaviour of their applications when executed on HTM. For instance, this tool can be used in the evaluation of the cost/benefit of each alternatives to improve application performance.

5.1 Tuning the Serialization Polices

Eigenbench was designed to study TM properties and thus does not account for changes in the parameters of the fall-back policy used in the HTM system to provide forward-progress guarantee. However, an interesting question is whether the parameters in these policies affect performance. For instance, what should be the maximum number of retries in the *SerControl* policy used for the experiments described in Section 4.2.2? To address this question we built htm-*pBuilder*, which acts as a wrapper over Eigenbench, to allow independent exploration of the parameters of a given fall-back policy. htm-*pBuilder* takes as input a set of values for transaction properties and parameters and tailors fall-back policies accordingly. The tool then automatically generates code for the various combinations of transaction properties and policy parameters, executes the code, and report results. The following sections presents new insights, gained from the experiments performed with htm-*pBuilder*, into the tuning of the *SerControl* policy.

5.2 The Effect on the Maximum Number of Retries

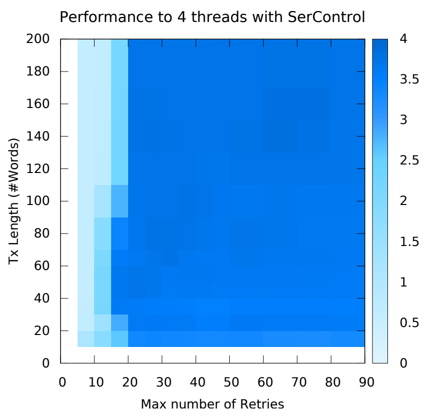
Section 4.2.2 described a systematic exploration of the characteristics of the Intel's TSX. That exploration set the TM properties to a baseline configuration and then varied a single TM property at a time. Those experiments allowed for a maximum of 20 retries in the *SerControl* policy. That value was obtained empirically as the best average result for a series of experiments with the RMS-TM [34] and STAMP benchmarks. Those experiments are repeated in this section, but this time we use the htm-*pBuilder* tool to vary the maximum number of retries. These new experiments revealed that in applications with low-to-moderate contention a higher threshold value results in better performance while in applications with high contention this threshold should remain close to 20.

For instance, the experiments reported in Figure 5.1 support this observation. For Figure 5.1(a) the working-set size is fixed in 32 Kbytes/thread and the write-ratio (pollution) in 10% — thus the contention level is low. The gradient plot reports the performance when the transaction length and the number of retries are varied. There is a significant performance change when the number of retries is greater than 20 for both small and large transactions. A comparison with Figure 4.1(a) on Section 4.2.2, where the number of retries was fix at 20, confirms that for that setting better performance is limited to transaction lengths between 30 and 60 words. For the experiment reported in Figure 5.1(b) the transaction length is fixed at 50 words and the write-ratio in 10%. The performance drop for 256 KBytes/thread when the maximum number of retries is equal 20, thus confirming the result in Figure 4.1(c) on Section 4.2.2. However the exploration with htm-*pBuilder* reveals that better performance can be obtained for larger work-set sizes if the maximum number of retries is increased. The exploration reported in Figure 5.1(d) showed that the write-ratio (or pollution) only affects the performance when the transaction length increase. The number of retries for this experiment was fixed in 20.

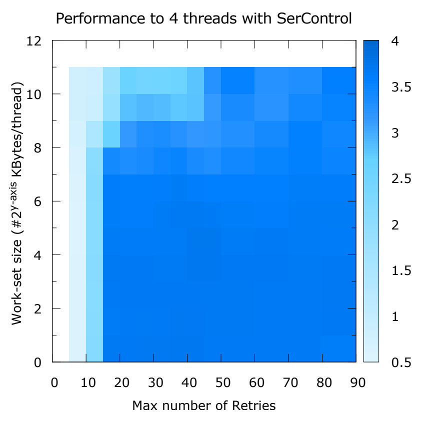
A similar experiment, not reported here, explored the performance for different levels

Figure 5.1: Two-dimensional Analysis with Eigenbench and *SerControl*

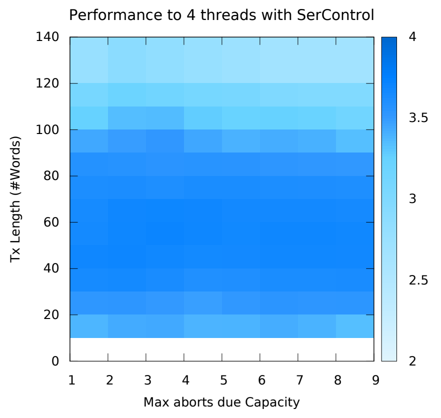
(a) Transaction Length X Retries



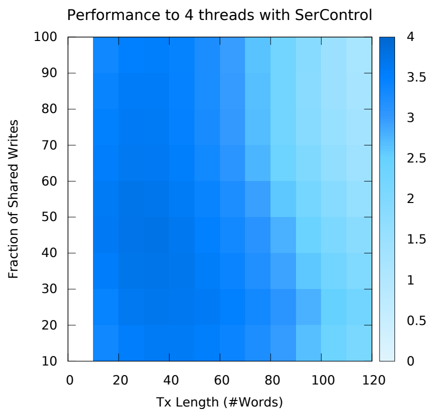
(b) Working-set size X Retries



(c) Transaction Length X Capacity



(d) Transaction Length X Pollution



of contention when the maximum number of retries varies. This experiment revealed that, when the contention is high, increasing the number of retries does not improve performance to the same extent as the low-contention case. This experimental result is most likely a consequence of the limited data cache line size in Intel's TSX. Thus, for high contention it is preferable to keep the retry threshold close to 20, or even less, to reduce the total number of aborts.

5.3 The Effect of Serialization on Capacity Aborts

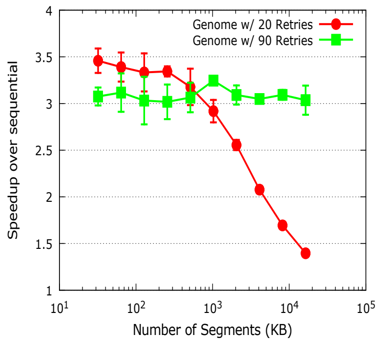
To prevent frequent aborts because of capacity overflow, the *SerControl* policy serializes a transaction that has suffered two consecutive capacity aborts. The ideia is that a capacity abort may be caused by competing transactions vying for the same storage resources. The htm-*pBuilder* facilitates an experimental evaluation of this strategy. Figure 5.1(c) shows the performance gradient when varying the transactional length and the number of consecutive capacity aborts allowed before the aborted transaction is serialized. The retry threshold utilized was 20, the working-set size was 32 Kbytes/thread and the pollution was 10%. These transaction properties allow comparisons with the results reported in Section 4.2.2. The experiment revealed that the strategy was correct but, perhaps, adjusting this value to 3 or 4 could be a better solution.

5.4 Applying new Findings in the STAMP Benchmark Suite

Section 5.2 reported the effect of using the maximum number of retries and section 5.2, the effect of exploring the number of capacity aborts allowed, before serialization. But these sections used a set of synthetic programs generated by Eigenbench. A practical question is whether the finding that a higher value for the maximum number of retries can be beneficial translates to actual benchmarks. To answer this question, this section applies those findings to the STAMP benchmarks.

This study varied the maximum number of retries from 20 to 90 in all STAMP benchmarks. The only benchmark where a significant change in performance is observed is Genome. The graph in Figure 4.3(e) in Section 4.2.4 indicates that the performance of genome is sensitive to transaction length. Figure 5.2 compares the speedup over sequential execution for the maximum number of retries equal 20 and 90. A value of 90 yields the best performance, but there is no significant difference for values above 70. The shape on the graph for threshold equals 90 shows that, although the performance had a small drop for the smallest number of segments, it maintains almost constant, regardless of the number of segments, confirming what was shown in the graph on Figure 5.1(a).

Figure 5.2: Study of maximum retries with STAMP



5.5 Avoiding Lemming Effect

The strategy adopted in the fall-back policy is to retry the execution of the transaction — with or without a time delay — to attempt to complete the transaction execution speculatively. In the face of persisting failure, a transaction must be completed by running it in a non-speculative execution mode. A common solution is to acquire a global lock to prevent other transactions from committing concurrently. However, the acquisition of a lock by a transaction causes every other transactions to abort. This can cause a chain effect, also known as the *lemming effect*, where the aborted transactions also try to acquire the lock [13].

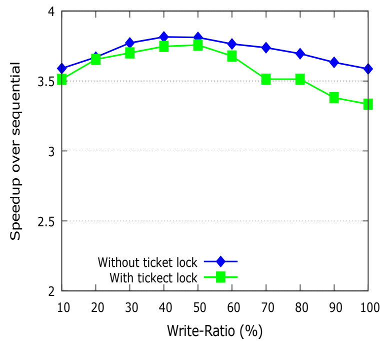
An alternative technique to the single global lock strategy is to use an auxiliary lock to prevent this lemming effect [2]. The idea is to guard the global lock acquisition with another lock. Aborted transactions have to acquire this auxiliary lock before serializing. This auxiliary lock is not added to the read set of transactions, thus avoiding the chain reaction effect.

In this experiment the auxiliary lock is a *ticket lock*. The *ticket lock* works as follows. Two memory locations - a *queue ticket* and a *dequeue ticket* — are accessed atomically. Initially both locations contain the value 0 to indicate that the ticket is not held. When a transaction needs to serialize its execution, it atomically reads and then increments the queue ticket. It then atomically compares the queue ticket that it read with the dequeue ticket's value. If they are the same, the transaction is permitted to try to obtain the global lock. If they are not the same, then another transaction must already be acquiring, or holds, the global lock and this transaction must busy wait or yield. When a transaction releases the global lock, it atomically increments the dequeue ticket thus allowing the next waiting transaction to acquire the global lock.

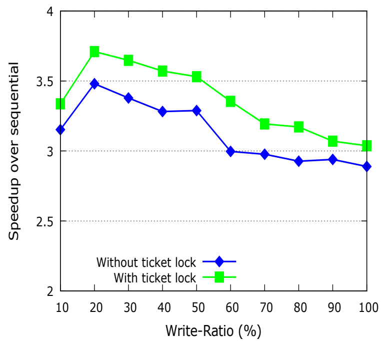
The experiment reported in Figure 5.3 used the *SerControl* policy with a maximum number of retries equals 20. For a transaction length of 30 words the overhead of the use of the ticket lock resulted in a performance degradation while with a larger transaction length of 70 words the ticket lock results in a small performance improvement over the standard single global lock solution.

Figure 5.3: Study of the use of ticket lock to mitigate lemming effect using Eigenbench

(a) Transaction Length = 30 words (120 bytes)



(b) Transaction Length = 70 words (280 bytes)



Chapter 6

Conclusions

The experimental evaluation on STM indicates that implementing better scheduling policies integrated with a dynamic transaction scheduler can yield significant performance benefits. This work describe the Best Alternative Transaction — BAT and Success-Rewarding Policy — SRP, two policies motivated by the observation that a global view of the transactions execution can improve accuracy of the scheduler's decision, thus minimizing aborts, specially for applications with longer executing transactions.

We have proposed and tested the BAT heuristic using two well-known and widely accepted benchmarks: STMBench7 and STAMP (not showed here) with TinySTM library as the baseline. We also did some tests with SwissTM and the preliminary results are well aligned with the results presented in this work. The results obtained with STMBench7, with emphasis on the Big and Huge data set sizes, show that BAT indeed improves the speedup of STMBench7 in setups that generated high contention (heavy workload), large number of threads, and with diverse but long duration transactions.

The Percentage of Effective Work — PEW metric combined with SRP and Dynamic Transaction Schedule — DTS, built atop of the TinySTM library, was also evaluated using STAMP and STMBench7. The experimental results indicate that DTS performs marginally better on most of then and is consistently better than the alternatives. The results obtained with STMBench7, with emphasis on the Big and Huge data set sizes, show that SRP indeed improves the speedup of STMBench7 in setups that generate high contention (heavy workloads), large number of threads, and with diverse- and long-duration transactions.

The results obtained with STAMP also confirm that the proposed approach points to the right direction because DTS does well for applications with higher transaction diversity, longer durations, and heavier workloads. For STMs the overhead of any associated mechanism is critical. The results indicate that it is possible to improve the performance of STMs using a transaction scheduler implemented outside the STM, with the combination of well-designed policies, provided the overheads of the associated mechanisms are kept under control. By combining DTS, SRP, and PEW the resulting system overcame the associated overheads and delivered non-trivial performance gains.

The experimental evaluations on HTM shows that the best-effort transactional memory provided by Intel's Haswell is simple and capable of improving performance over a variety of workloads. However, performance can depend strongly on the software support systems. While transaction footprint and working-set size constraints dictate the range of effective transactions, choices made in the lock based fallback policy can considerably affect performance, especially when capacity-limited transactions are executed. The dynamic nature of the cache means that the capacity-aborted signal is not a reliable indicator that a transaction will not complete. This observation is supported by the success of the *SerControl* fallback policy, which allows transactions that suffer capacity aborts to retry.

Transactional memory is a natural fit for multi-core architectures, but the success of transactional memory will be partially determined by the quality of early implementations. A unique evaluation of transaction footprint and working-set size through input modification of the STAMP benchmarks shows that the best-effort nature and capacity limitations of Intel's HTM underscores that TM is not a parallel synchronization solution to all applications.

Bibliography

[1] A. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft Specification of Transactional Language Constructs for C/C++. In Intel, 2012.

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

[3] Mohammad Ansari, Mikel Lujan, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-Abort: Improving transactional memory performance through dynamic transaction reordering. In *High Performance Embedded Architectures and Compilers (HIPEAC)*, pages 4–18, Paphos, CYPRUS, January 2009.

[4] Ehsan Atoofian. Speculative contention avoidance in software transactional memory. In *International Symposium on Parallel and Distributed Processing*, pages 1417–1423, Phoenix, Arizona, USA, September 2011. IEEE.

[5] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *Symposium on Principles of Distributed Computing (PODC)*, pages 308–315, Denver, Colorado, USA, July 2006.

[6] Hagit Attiya and Alessia Milani. Transactional scheduling for read-dominated workloads. *Journal of Parallel and Distributed Computing (JPDC)*, 72(10):1386–1396, October 2012.

[7] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Proactive transaction scheduling for contention management. In *International Symposium on Microarchitecture (MICRO)*, pages 156–167, New York, NY, USA, December 2009.

[8] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional mem-

ory. In *International Symposium on Computer Architecture (ISCA)*, pages 81–91, June 2007.

[9] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 227–238. IEEE Computer Society, June 2006.

[10] Dongmin Choi, Seung Hun Kim, and Won W. Ro. Conflict avoidance scheduling using grouping list for transactional memory. In *International Symposium on Parallel and Distributed Processing Workshop*, IPDPSW '12, pages 547–556, Washington, DC, USA, 2012. IEEE Computer Society.

[11] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Riviere. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *European Conference on Computer Systems (EUROSYS)*, pages 27–40, Paris, France, April 2010.

[12] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. ASF: AMD64 extension for lock-free data structures and transactional memory. In *International Symposium on Microarchitecture (MICRO)*, pages 39–50, Atlanta, Georgia, USA, December 2010.

[13] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. Applications of the adaptive transactional memory test platform. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, feb 2008.

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

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

[16] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *Symposium on Principles of Distributed Computing (PODC)*, pages 125–134, Toronto, Canada, August 2008.

[17] Aleksandar Dragojevic, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *Symposium on Principles of Distributed Computing (PODC)*, pages 7–16, Calgary, Alberta, Canada, August 2009.

[18] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

[19] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 237–246, Salt Lake City, Utah, USA, February 2008.

[20] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *International Symposium on Distributed Computing (DISC)*, DISC'09, pages 93–107, Berlin, Heidelberg, 2009. Springer-Verlag.

[21] Bhavishya Goel, Ruben Titos Gil, Anurag Negi, Sally A McKee, and Per Stenström. Performance and energy analysis of the restricted transactional memory implementation on haswell. In *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS*, pages 615–624, 2014.

[22] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *International Symposium on Distributed Computing (DISC)*, pages 303–323, Cracow, Poland, September 2005.

[23] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Symposium on Principles of Distributed Computing (PODC)*, pages 258–264, Las Vegas, Nevada, USA, July 2005.

[24] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A benchmark for software transactional memory. In *European Conference on Computer Systems (EUROSYS)*, pages 315–324, Lisbon, Portugal, March 2007.

[25] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *International Symposium on Computer Architecture (ISCA)*, pages 102–113, München, Germany, June 2004. IEEE Computer Society.

[26] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory.* Morgan & Claypool Publishers, 2 edition, June 2010.

[27] Tomer Heber, Danny Hendler, and Adi Suissa. On the impact of serializing contention management on STM Performance. *Journal of Parallel and Distributed Computing (JPDC)*, 72(6):739–750, June 2012.

[28] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, Boston, Massachusetts, USA, July 2003.

[29] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *International Symposium on Computer Architecture (ISCA)*, pages 289–300, San Diego, California, USA, 1993. ACM.

[30] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *The IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–11, December 2010.

[31] IBM Corporation. Performance optimization and tuning techniques for ibm processors, including IBM POWER8. `http://www.redbooks.ibm.com/abstracts/sg248171.html`, November 2014.

[32] IBM Research. Patching and building HTM-enabled STAMP. `http://researcher.ibm.com/researcher/view_person_subpage.php?id=5820`, 2014.

[33] Intel Corporation. Chapter 12: Intel's Transactional Synchronization Extensions (TSX) Recommendations. `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`, July 2013.

[34] Gokcen Kestor, Srdjan Stipic, Osman Unsal, Adrian Cristal, and Mateo Valero. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *The ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, Salt Lake City, Utah, USA, February 2009.

[35] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Symposium on Principles and*

Practice of Parallel Programming (PPoPP), pages 79–90, Bangalore, India, January 2010.

[36] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *The IEEE International Symposium on Workload Characterization (IISWC)*, pages 35–46, Seattle, WA, USA, September 2008.

[37] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *International Symposium on High-Performance Computer Architecture*, pages 254–265, Austin, Texas, USA, February 2006. IEEE Computer Society.

[38] Daniel Nicácio, Alexandro Baldassin, and Guido Araújo. LUTS: A lightweight user-level transaction scheduler. In *International Conference on Algorithms And Architectures For Parallel Processing*, pages 144–157, Mellbourne, Australia, October 2011.

[39] William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Symposium on Principles of Distributed Computing (PODC)*, pages 240–248, Las Vegas, Nevada, USA, 2005.

[40] Martin Schindewolf, Barna Bihari, John Gyllenhaal, Martin Schulz, Amy Wang, and Wolfgang Karl. What scientific applications can benefit from hardware transactional memory? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 90:1–90:11, Los Alamitos, CA, USA, 2012.

[41] Gokarna Sharma, Brett Estrade, and Costas Busch. Window-based greedy contention management for transactional memory. In *International Symposium on Distributed Computing (DISC)*, pages 64–78, Cambridge, Massachusetts, USA, September 2010.

[42] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996.

[43] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 141–150, Raleigh, North Carolina, USA, February 2009.

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

[45] Mike Dai Wang, Mihai Burcea, Linghan Li, Sahel Sharifymoghaddam, Greg Steffan, and Cristiana Amza. Exploring the performance and programmability design space of hardware transactional memory. In *The ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, Raleigh, NC, USA, March 2014.

[46] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *International Symposium on High-Performance Computer Architecture*, pages 261–272, Phoenix, Arizona, USA, February 2007.

[47] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013.

[48] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 169–178, Munich, Germany, June 2008.

Appendix A

IBM Power8 Evaluation

This appendix shows the preliminary results of the characteristics and hardware constraints presented by IBM's Power ISA TM Extensions included in the Power8 machine. The IBM's Power8 processor architecture comes with up to 12 cores per chip. It uses Simultaneous Multithreading (SMT) to provide multiple streams of hardware execution. Power8 provides eight SMT hardware threads per core and can be configured to run in SMT8, SMT4, SMT2, or single-threaded mode. However, there are some performance tradeoffs. When you are in SMT mode, there is a trade-off between overall CPU throughput and the performance of each hardware thread. SMT allows multiple instruction streams to be run simultaneously, but this concurrency can cause some resource conflict between the instruction streams. This conflict can result in a decrease in performance for an individual thread, but an increase in overall throughput.

Power8 systems use 128-byte length cache lines. Compared to Intel Haswell processors (64-byte cache lines), these larger cache lines have the advantage of increasing the possible reach with the same size cache directory, and the efficiency of the cache by covering up to 128-bytes of hot data in a single line. However, it also has the implication of potentially bringing more data into the cache than needed for fine-grained accesses (that is, less than 64 bytes). Another advantage of the fact that Power8 has a larger cache line is that it can be best benefit from locality (both temporal and spatial). Locality arises from simple and natural program structures. For example, most programs contain loops, so the data are likely to be accessed repeatedly, showing high amounts of temporal locality. Access to data also exhibit a natural spatial locality. For example, access to elements of an array will naturally have high degrees of spatial locality.

We conducted the performance evaluation on a Power8 machine with CPU op-mode 64-bit (that is, the word equals 8 bytes), 16 sockets, 1 core per socket and 8 threads per core. The machine also contains 3 NUMA nodes. However, the experiment used only 4 threads to do a (not so fair) comparison, despite the difference in its architecture with respect to Intel's TSX used in our experiments. Threads were pinned to different sockets and NUMA auto balancing was turned off.

The results presented in all experiments are averaged over twenty executions and the graphs show a 95% confidence interval. We conducted the following experiments with the htm-*pBuilder*. The graphs in Figure A.1 display the trends for transaction length and working-set size using *MaxRetry* and *SerControl* policies configured with maximum number of retries equals twenty. The experiment reported on Figure A.1 demonstrates that Power8 is more limited with respect to the transaction footprint then Intel's TSX experiments showed. In these experiments we used the write ratio equals 10%. The graph on Figure A.1(a) shows the performance with the variations on transaction length while the graphs on Figure A.1(b) and Figure A.1(c)) show the outcomes for abort fraction over commits and capacity-abort fraction over commits, respectively. The graph in Figure A.1(d) shows the effect of different working-set sizes for transaction length equals to 10 words. Contrary to Intel's TSX, which shows a dramatic speedup drop when the capacity of L2 cache is exceeded (see the graph of Figure 4.1(c), on Section 4.2.2) the behaviour maintains the same for the Power8. To better understand this new behaviour, experiments that stress the Power8 cache sizes are necessary. Again, the graphs on Figures A.1(e) and A.1(f) points out that *SerControl* is dramatically more reluctant to serialize when capacity aborts occur, confirming one of the payoffs of the *SerControl* policy.

Finally, to see the effect of the variation on the maximum number of retries, the graphs on Figure A.2 show the performance when transaction length varies with the maximum number of retries. One can note that the behaviour of *MaxRetry* policy is not affected with the variation on the maximum number of retries while the performance of *SerControl* improves at it increases from fifteen (showed on Figure A.2(a)) to twenty, 20 (Figure A.2(b)) and to twenty-five (Figure A.2(c)).

Figure A.1: Analysis of Power8 TM using htm-*pBuilder*

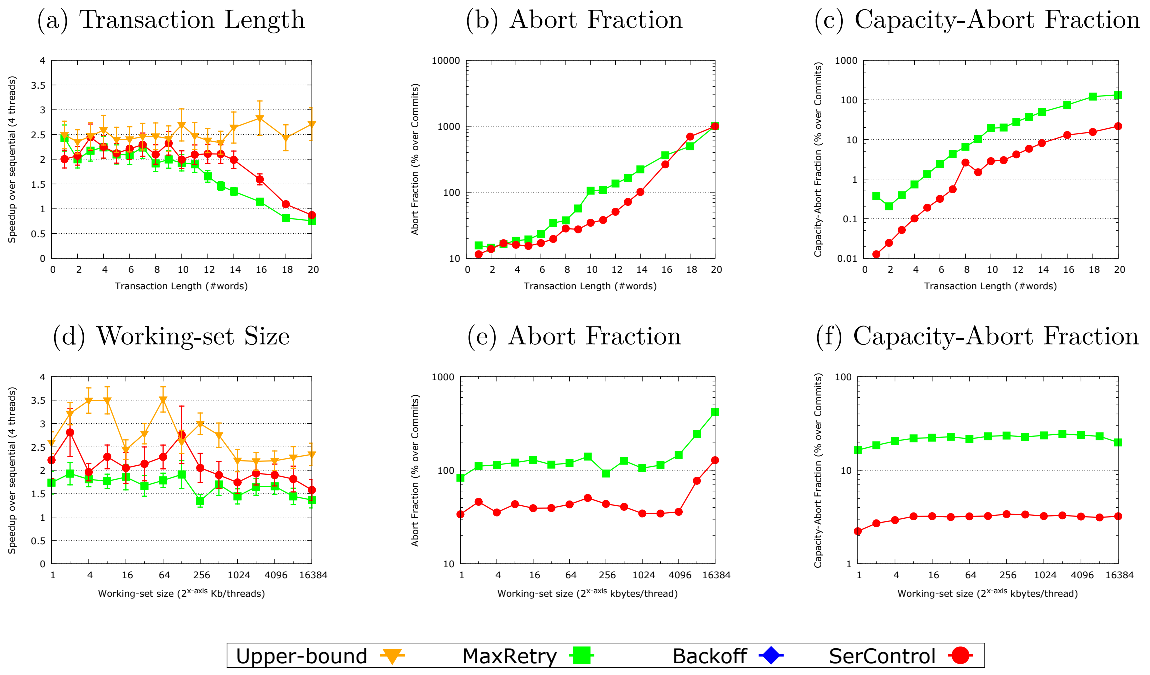
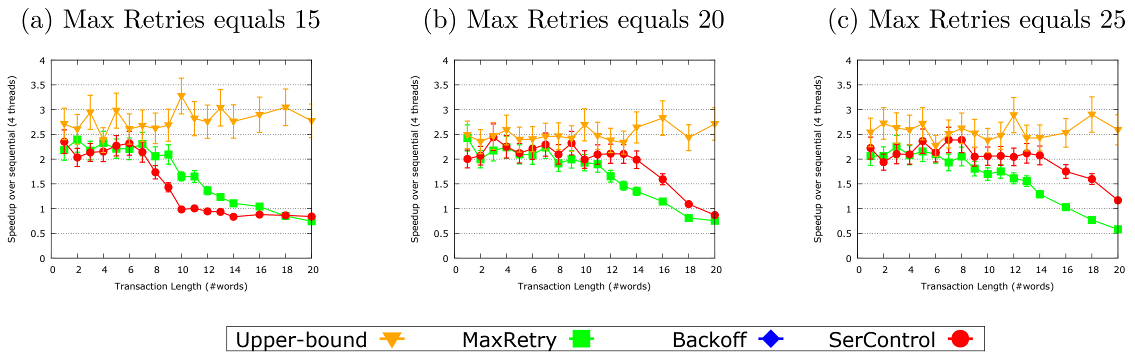


Figure A.2: The effect on the Maximum Number of Retries on Power8



Appendix B

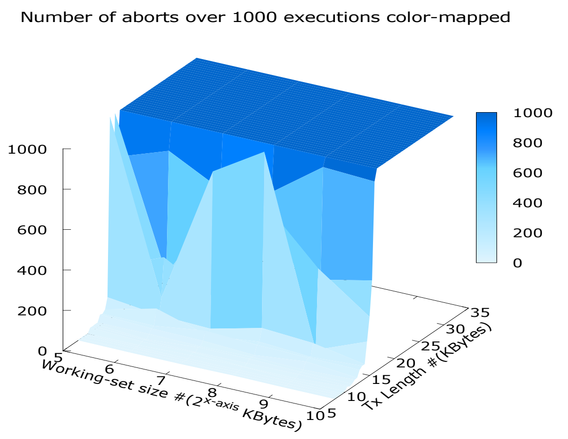
Haswell Processor Characteristics

According to Intel, the Haswell processor has equipped with L1 data cache (L1D) per core of 32KBytes, 8-way associative and write-back policy. The L2 cache is a 256KB, 8-way associative and write-back design with ECC protection. The L2 is neither inclusive nor exclusive of the L1 data cache. Haswell's transactional memory retains the write-set and read-set of a transaction in the L1 data cache. The L2 cache is non-transactional, so if a write-set cache line is evicted, the transaction will abort. It appears that in some circumstances read-set lines can be safely evicted from the L1 and are tracked using another hardware mechanism. One possibility is a small on-chip transactional victim buffer, or some sort of storage in memory (e.g., bloom filter). In the case of an abort, all the write-set lines are flushed from the L1D, while a commit will make all write-set lines atomically visible. Cache-based systems have potential limits due to associativity, and certain access patterns will cause transaction aborts due to cache contention. The caches, TLBs and fill buffers are competitively shared by any active threads, however, the load and store buffers are statically partitioned. Store operations only need to write the address (and eventually data) into the store buffer. In contrast, load operations must write into the load buffer and also probe the store buffer to check for any forwarding or conflicts.

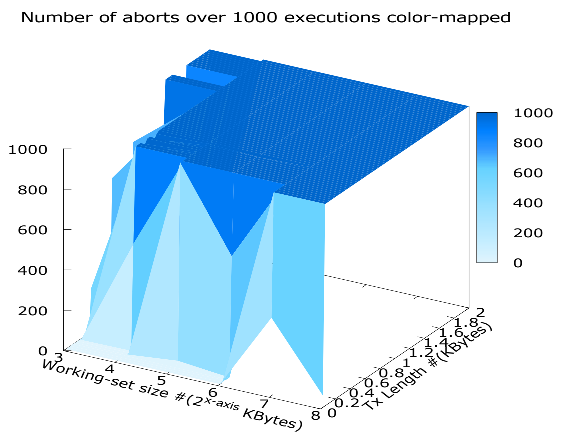
Intel Transactional Synchronizations Extensions — Intel's TSX, is a recent addition to the Intel architecture that provides programmers with hardware transactional memory in the Haswell processor. We did a series of experiments to determine the primary characteristics of TSX. The first experiment aims to determine how speculative states Intel's TSX can sustain on the same set before they cause an abort due the limitations in cache store capacity. One static transaction was drawn to never ends. The result show that this only begins to occur with 8 threads (or 8 speculative states) but the most important observation in the experiment shows that (a) TSX uses eager-conflict detection, and (b) the cache is not multi-versioned.

Another experiment that demonstrate the limitations of TSX is showed in the Figure B.1. The graph on the Figure B.1a , the speculative writes are doing in a sequential way (y-axis) over the array which size is giving by x-axis. In the graph on the Figure B.1b, the writes are doing in a random way (y-axis) over the array which size is giving by x-axis. One can noted that in this second experiment, the L1 cache is exhausted very fast. This is because in the first experiment, the sequential write fills the cache lines until their exhaust while in the second experiment the cache line is filled by useless data due to cache protocol.

A final experiment comparing the results of both read from and write to on array was conducted to confirm that TSX uses a separate mechanism, perhaps a Bloom Filters, to track speculative reads that have been evicted from the L1 data cache.



(a) Speculative Sequential Writes



(b) Speculative Random Writes

Figure B.1: Characterization of TSX

Appendix C

htm-*pBuilder* Configuration

The Figure C.1 shows the block diagram of the htm-*pBuilder*. Its configuration file can be conceptually divided into 3 distinct parts. In the first part, one can specify the characteristics of the target machine as the size of data cache and the cache line (L1), the size of one word (in number of bytes), the number of available cores and processors. In the second part, one can choose one of the three preconfigured policies (*MaxRetry, Backoff* or *SerControl*, to be used in the experiments as well as the tuning that can be done in selected policy, or yet, customize a new policy, following some pre-established criteria to allow, for instance, the benchmark to properly collect statistics information during its execution. The idea is that the tool can be used for different architectures and policies. Today, the input files are customized for TSX extensions of Intel Haswell and Power ISA extensions of IBM Power8 [31].

In the third part, the experimenter can specify the type of data structures (e.g., array, linked-list, red-black tree) as well the initial size of each data structure (*i.e.*, number of elements). To better reproduce the behaviour and pathologies found in real TM applications, and similar to Eigenbench, the experimenter can choose at least one of three kinds of data: *shared, private* and *exclusive*. The *shared data* are shared among all threads and accessed transactionally, to enable the experimenter to control the level of contention between transactions. With the *private data* each thread has its own private data, but this data is also accessed transactionally there is competition for the transactional resources (e.g., speculative states) as well as the probability of the occurrence of false-sharing. The *exclusive data* are also private to each thread, but in this case, the data is accessed only outside of the transactions. This kind of data help to analyse the coexistence between transactional and non-transactional code and evaluate the density characteristic of TM applications, defined by the ratio of shared access (shared instructions) to the total number of instructions (shared and non-shared) inside transactions.

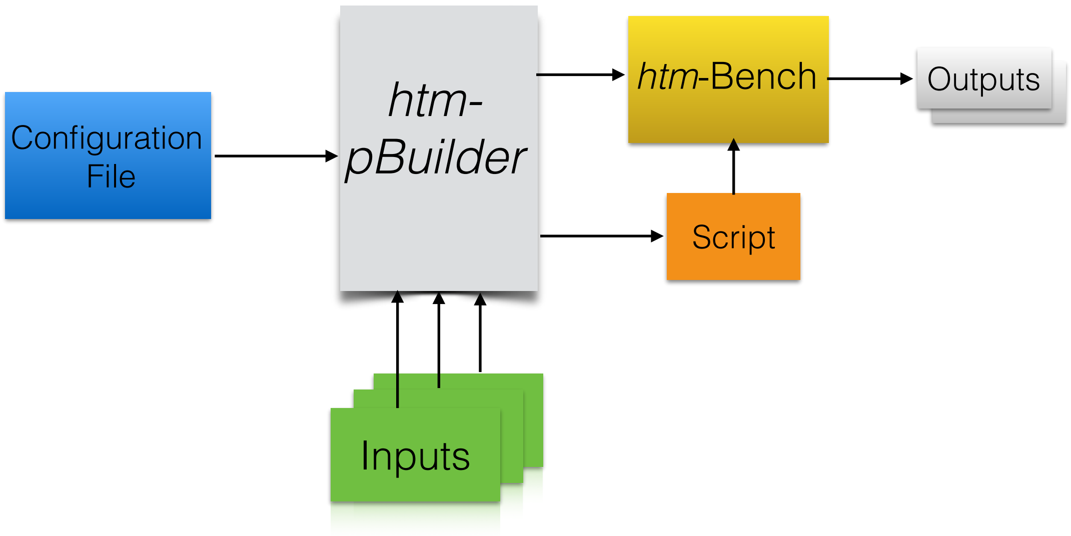


Figure C.1: Schematic view of htm-*pBuilder*.

After choosing the data structures, the experimenter can select the transaction length, pollution (i.e., write-ratio), and temporal locality, etc., to explore the TM characteristics and to evaluate the boundaries of the application space of a HTM machine.

C.1 Multi-Dimensional Configuration examples

This section presents some configuration files that were used in the experiments reported in Section 5.2 and Section 5.3.

The example below was used in the experiment “Transaction Length versus Number of Retries” reported on the Graph of Figure 5.1a. This experiment used one type of transaction executed by four threads in the Intel Haswell machine configured with 4 cores. The tool does not pin threads to cores (parameter `Affinity: no`). The data type used in the experiments is a private array of 8192 elements (or 32768 bytes) per thread (parameters `DS2` and `IS2`). The transaction footprint is determined by transaction length (parameter `LEN2`) and write-ratio (parameter `POL2`). In the example, `POL2` equals 10%. The tool automatically generates code and executes the generated code 10 times (parameter `Executions: 10`) for each combination of maximum number of retries (specified by `_MAX_RETRY` parameter) and transaction length (parameter `LEN2`). The parameter `Loops: 1000000` controls the duration of the execution of each transaction. The tool captures the elapsed times and outcomes (e.g., number of aborts, number of commits) and calculates the average and 95% confidence interval to report the results.

```
#------------------------
# Machine Characteristics
# -----------------------
Brand: Intel
Cores: 4
Processors: 8
L1S: 64
Affinity: No

#------------------------
# Policy Specification
#------------------------
Policy: SerControl
Ticket: No
Lock: yield
_MAX_RETRY: (5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80, 90)

#----------------------------
# Global Execution Parameters
#----------------------------
DS2: Array
IS2: 8192
LEN2: (10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140,
150, 160, 170, 180, 200)
POL2: 10
Threads: 4
Transactions: 1
```

```
Loops: 1000000
Executions: 10
```

The next configuration file was used in the experiment “Working-set size versus Number of Retries” reported on Graph of Figure 5.1b. This experiment fixes the transaction length in 50 words and makes pollution equals 10. The omitted parameters use their default values.

```
#------------------------
# Machine Characteristics
# -----------------------
Brand: Intel
Affinity: No

#------------------------
# Policy Specification
#------------------------
Policy: SerControl
Ticket: No
Lock: yield
_MAX_RETRY: (5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80, 90)

#----------------------------
# Global Execution Parameters
#----------------------------
DS2: Array
IS2: (1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144,
      524288, 1048576, 2097152)
LEN2: 50
POL2: 10
Threads: 4
Transactions: 1
```

In the last example below, we varied the combination of transaction length and maximum number of capacity aborts before the policy decides to serialize. The result of this experiment: “Transaction Length versus Max aborts due Capacity Overflow”, appears in Figure 5.1c.

```
#------------------------
```

```
# Machine Characteristics
# ------------------------
Brand: Intel

#------------------------
# Policy Specification
#------------------------
Policy: SerControl
_MAX_RETRY: 20
_OVERFLOW_CAPACITY: (1,2,3,4,5,6,7,8)

#----------------------------
# Global Execution Parameters
#----------------------------
DS2: Array
IS2: 8192
LEN2: (10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 140)
POL2: 10
```

C.2 Customizing the Delay function

Policies such as *Backoff* and *SerControl* (in one of its three actions) use a time delay before restarting an aborted transaction. The delay duration in both policies is chosen uniformly at random from a range whose size increases exponentially with every restart. With htm-*pBuilder*, the experimenter can specify a different function to choose the delay duration. There are two ways to accomplish this. The first one, simple but limited in terms of alternatives, is by using the `inline` atribute. The *Backoff* policy (and also the *SerControl*) has two pre-defined constants: `_MIN_BACKOFF` and `_MAX_BACKOFF` with default value of 4 and 13, respectively, can be modified. Also, two local variables: `_backoff` and `_wait` that can be used in the `Inline` attribute. They are initialized in policies such as this:

```
int _backoff = _MIN_BACKOFF;
unsigned long long int _wait = 0;
```

The desired delay function will be inserted in policies between the code below:

```
...
```

```
  if (_status == BACKOFF) {
   /*  the delay function will be inserted here */
    _backoff++;
  }
  ...
```

Following is an example of customized delay function using the `Inline` attribute. In this example, the `usleep()` function was used. This function suspends the thread execution for microsecond intervals specified by its parameter.

```
# Policy Specifications
Policy: Backoff
Retries: 20
Delay: Custom
_MIN_BACKOFF: 1
_MAX_BACKOFF: 6
Inline: (
   if (_backoff > _MAX_BACKOFF) _backoff = _MIN_BACKOFF;
   _wait = 1ULL << _backoff;
   usleep (_wait);
   _backoff++;
)
```

Another possibility to customize the delay function is to define an external `C` function. The example below, uses the instruction `PAUSE`, that consume 38-40 clocks.

```
//--------------
// file delay.h
//--------------

#ifndef DELAY_H
#define DELAY_H

#include <immintrin.h>

static inline void _my_delay(int* _backoff) {
  unsigned long long int _my_wait = 1ULL << _backoff;
  while (_my_wait--) _mm_pause();
}

#endif
```

Next the reference to the header file must be included in the configuration file:

```
# Policy Specifications
Policy: Backoff
Retries: 20
Delay: Custom
Include: "delay.h"
Inline: (
   _my_delay (&_backoff);
   _backoff++;
)
```

C.3 Reproducing the Performance Behaviour of Applications

Applications usually have a mix of various transaction types and exhibit complex memory access patterns. One of the powers of Eigenbench, and naturally extended to htm-*pBuilder*, is to mimic real applications by measuring the appropriate TM values and then mapping these values to eigen-characteristics. This is normally obtained via instrumentation-based profilling or simulation and yet, through analysis of the source code.

One goal of this feature is to predict the performance of an application, originally synchronized using with locks when it is re-written to use TM for synchronization. Moreover, this mimic can help answer questions like: (a) whether it is preferable to convert multiple lock acquisitions or critical sections into a single transactional region; or (b) if it is better amortize the overhead necessary to execute a transaction; or (c) whether it is preferable to maintain transactions with smaller footprint to reduce wasted work.

Another feature that can be developed for htm-*pBuilder* is to estimate the performance impact of the proposed modifications to existent TM application without needing to prototype, thus reducing the cost of the estimation. The first step is to mimic the existing TM application. It is possible to automate this step through instrumentation of the fall-back policy. The application is executed with the instrumented policy and guided by scripts generated by htm-*pBuilder*. For instance, the fall-back policy can be extended to collect the size, measured in number of clock cycles, of dynamic transactions that commit and scripts clustering the result data can be used to estimate the different types of static transactions. The same idea can be used to collect the write-ratio and locality, counting the number and frequency of different locations (or cache lines) touched by the transaction. The imitating application can then be run to measure the current performance and, after that, change the parameters to reflect the proposed changes and execute again the imitating application to estimate the impact on performance.