UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

SILVIO RICARDO CORDEIRO

Code Profiling and Optimization in Transactional Memory Systems

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. Nicolas Maillard
Advisor

Porto Alegre, February 2014

CIP – CATALOGING-IN-PUBLICATION

Cordeiro, Silvio Ricardo

Code Profiling and Optimization in Transactional Memory Systems / Silvio Ricardo Cordeiro. – 2014.

77 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2014.

1. Transactional Memory. 2. Profiling. 3. Scheduling. 4. Shared Memory. 5. Parallel Programming. 6. High-Performance Computing. I. Maillard, Nicolas. II. Título.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Nicolas Maillard, and my unofficial co-advisor José Nelson Amaral for their interest in my pursuit of this work.

I would also extend that gratitude to my colleagues at the GPPD research group, in particular Stéfano, with whom I shared many hours of (sometimes pointless) discussions about anything ranging from theoretical computer science to parallel programming to obscure semantics of programming languages and even to whether $\sum_{n=0}^{n}$ should have any meaning at all — jury's still out on that one.

I would also like to thank my family, as well as everyone else who has been a part of my personal life, in particular Jonier and Juliana.

Profiling e Otimização de Código
em Sistemas de Memória Transacional

RESUMO

Memória Transacional tem se demonstrado um paradigma promissor na implementação de aplicações concorrentes sob memória compartilhada que busquem evitar um modelo de sincronização baseado em *locks*. Em vez de sujeitar a execução a um acesso exclusivo com base no valor de um *lock* que é compartilhado por threads concorrentes, uma aplicação sob Memória Transacional tenta executar seções críticas de modo otimista, desfazendo as modificações no caso de um conflito de acesso à memória. Entretanto, apesar de a abordagem baseada em *locks* ter adquirido um número significativo de ferramentas automatizadas para a depuração, *profiling* e otimização automatizados (por ser uma das técnicas de sincronização mais antigas e mais bem pesquisadas), o campo da Memória Transacional ainda é comparativamente recente, e programadores frequentemente precisam adaptar manualmente suas aplicações transacionais ao encontrar problemas de eficiência.

Este trabalho propõe um sistema no qual o *profiling* de código em uma implementação de Memória Transacional simulada é utilizado para caracterizar uma aplicação transacional, formando a base para uma parametrização automatizada do respectivo sistema especulativo para uma execução eficiente do código em questão. Também é proposta uma abordagem de escalonamento de threads guiado por *profiling* em uma implementação de Memória Transacional baseada em software, usando dados coletados pelo *profiler* para prever a probabilidade de conflitos e determinar que thread escalonar com base nesta previsão. São apresentados os resultados de experimentos sob ambas as abordagens.

Palavras-chave: Memória Transacional, Profiling, Escalonamento, Memória Compartilhada, Programação Paralela, Processamento de Alto Desempenho.

Code Profiling and Optimization in Transactional Memory Systems

ABSTRACT

Transactional Memory has shown itself to be a promising paradigm for the implementation of shared-memory concurrent applications that eschew a lock-based model of data synchronization. Rather than conditioning exclusive access on the value of a lock that is shared across concurrent threads, Transactional Memory attempts to execute critical sections optimistically, rolling back the modifications in the event of a data access conflict. However, while the lock-based approach has acquired a significant body of debugging, profiling and automated optimization tools (as one of the oldest and most researched synchronization techniques), the field of Transactional Memory is still comparably recent, and programmers are usually tasked with an unguided manual tuning of their transactional applications when facing efficiency problems.

We propose a system in which code profiling in a simulated hardware implementation of Transactional Memory is used to characterize a transactional application, which forms the basis for the automated tuning of the underlying speculative system for the efficient execution of that particular application. We also propose a profile-guided approach to the scheduling of threads in a software-based implementation of Transactional Memory, using collected data to predict the likelihood of conflicts and determine what thread to schedule based on this prediction. We present the results achieved under both designs.

Keywords: Transactional Memory, Profiling, Scheduling, Shared Memory, Parallel Programming, High-Performance Computing.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACID	Atomicity, Consistency, Isolation and Durability
AMD	Advanced Micro Devices
API	Application Programming Interface
ASF	Advanced Synchronization Facility
ATMTP	Adaptive Transactional Memory Test Platform
CAS	Compare-and-Swap
CILUTS	Conflict Intensity LUTS
CPU	Central Processing Unit
ECR	Execution Context Record
GCC	GNU Compiler Collection
GEMS	General Execution-driven Multiprocessor Simulator
GNU	GNU is Not Unix
GPPD	*Grupo de Processamento Paralelo e Distribuído*
HPC	High Performance Computing
HTM	Hardware Transactional Memory
IBM	International Business Machines Corporation
ICC	Intel C++ Compiler
JSON	JavaScript Object Notation
LL/SC	Load-Linked/Store-Conditional
LUTS	Lightweight User-level Transaction Scheduler
MESI	Modified/Exclusive/Shared/Invalid protocol
NAS	NASA Advanced Supercomputing
ORC	Open Research Compiler

OpenMP	Open Multi-Processing
PARSEC	Princeton Application Repository for Shared-Memory Computers
PID	Process ID
POSIX	Portable Operating System Interface for Unix
RISC	Reduced Instruction Set Computing
SESC	Superscalar Simulator
SPEC	Standard Performance Evaluation Corporation
STAMP	Stanford Transactional Applications for Multi-processing
STM	Software Transactional Memory
SUIF	Stanford University Intermediate Format
TL2	Transactional Locking II
TLB	Translation Look-aside Buffer
TM	Transactional Memory
TxOS	Transactional Operating System
UFRGS	*Universidade Federal do Rio Grande do Sul*
XMESI	Transactional MESI

CONTENTS

1 INTRODUCTION

Programmers have always struggled with the implementation of parallel code that is both efficient and scalable, while maintaining its correctness. Due to a constant-rate improvement in single-processor performance that lasted for decades, concurrent programing was traditionally seen as an area that is limited to high-performance scientific applications, where it is crucial that execution time be minimized (RAUBER; RÜNGER, 2010).

In the last decade, however, the hardware industry has reached the physical limits of heat dissipation in a processor chip. In order to maintain the performance improvements, computer architects decided that the best solution was to replace that single-core processor design with an architecture that would be able to sustain multiple energy-efficient cores in a single chip (ASANOVIC et al., 2009).

The arrival of these multicore processors has presented a new challenge to the programming industry, in that now an application can only benefit from new hardware when it has been explicitly programmed so that its workload is divided among the available execution units (RAUBER; RÜNGER, 2010). Applications that do not take the extra work of distributing workload and synchronizing the memory access across processor cores will not be able to take advantage of new hardware anymore.

Due to the possibility of having multiple flows of execution competing for memory resources, multicore processors offer hardware instructions that can be used to implement higher-level software techniques to allow programmers to determine which sections of code should be protected, in order to guarantee mutual exclusion during data access among threads (HENNESSY; PATTERSON, 2006).

The most common solution that is adopted to avoid conflict between concurrent operations in a shared-memory environment is lock-based synchronization. However, in spite of how widespread it is, lock-based programming has some intrinsic deficiencies which makes it difficult for programmers to write scalable and efficient code.

A lock-based synchronization technique requires that programmers define critical sections to protect the concurrently executing instructions. Moreover, performance considerations require that parallel programmers choose the ideal locking granularity taking into account the costs associated with fine-grained locking — all the while taking care not to introduce deadlocks (HARRIS et al., 2005).

When implementing parallel programs through lock-based synchronization, one of the frequent source of problems that is identified is the lack of *composability* (LARUS; RAJWAR, 2006). While two sequentially correct sections of code can always be combined to form a correct sequential algorithm, the equivalent is not true for lock-based code executing in parallel, where the details of the underlying smaller components cannot be abstracted away from the combined whole.

One frequently cited example is that of a data structure with two thread-safe methods: *insert element* and *remove element*. Given two instances *A* and *B*, one wishes to compose these operations into a thread-safe *move element* subroutine that removes an element from *A* and inserts it into *B*. If the relevant methods depend on locks as a way to guarantee thread-safety, the only way to compose these operations is by acquiring an internal lock to both instances before the execution of *move element*. If another threads tried to hold the same locks in a different order, they might both come to a deadlock — even though both threads could have correctly finished execution if they ran sequentially (HARRIS et al., 2005).

In spite of all the difficulties in parallel code implementation, database systems have been successfully executing queries in parallel for a long time now. The key factor that has contributed to this result is the fact that the programming model for queries is centered around the notion of a *transaction*, which is a computation that must be entirely executed as if it was the only operation accessing the underlying database, in isolation from concurrently executing transactions.

As a way of solving the problem of parallel code composition, one of the synchronization techniques that has been researched during the last two decades is the Transactional Memory (TM). This synchronization model applies the concept of typically disk-based database transactions to a more memory-based programming environment (HARRIS; LARUS; RAJWAR, 2010). A transaction, in this context, consists of a sequence of operations that is seen as indivisible by any other thread of execution. It can either execute completely or fail and restore the modified values in memory back to their previous state.

Transactional Memory differs from from traditional approaches that rely on mutual exclusion by the fact the underlying system is allowed to optimistically execute a set of transactions concurrently, aborting some of them if any conflict with another transaction is detected during execution time (DICE et al., 2009; BAUGH; NEELAKANTAM; ZILLES, 2008). In case the execution of a transaction is aborted due to conflict, it needs to be executed again.

While this concept of a transactional memory was proposed as early as 1986 (KNIGHT, 1986), it did not receive much attention until it was shown possible through the definition of a set of machine-language instructions that followed the appropriate semantics (HERLIHY; MOSS, 1993). Soon after that, research sparked from the initial context of multiprocessor architectures and into software implementations, lead by the works of (SHAVIT; TOUITOU, 1995).

There have been many Hardware Transactional Memory proposals over the years, but only recently have we seen real implementations. The 2008 Rock processor was the first commercial processor to include Transactional Memory support, but in spite of the good early results (DICE et al., 2009), the project was canceled before its release. In 2012, it was followed by IBM's Blue Gene/Q processor (WANG et al., 2012), as well as Intel's specification for transactional support on their then-upcoming Haswell architecture (Intel Corporation, 2012).

Research on Software Transactional Memory has been going on in parallel to the hardware research, and they have obtained considerably distinct results. Differently from hardware implementations of transactions, that are physically constrained to a given amount of inner buffer memory and just fail when there is a capacity overflow, software implementations are able to handle any size of transaction as long as there is enough main memory available. Initial implementations were very slow, in spite of their high scalability. Over time, more efficient algorithms have been proposed (DICE; SHALEV; SHAVIT,

2006), but it is still questionable whether software implementations could ever be efficient without specific hardware support (CASCAVAL et al., 2008).

As an attempt to tackle the deficiencies commonly perceived in both hardware and software-implemented Transactional Memory systems, there has been some research on Hybrid Transactional Memory, which tries to use a hardware-only implementation as much as possible, but relegates to the software-based approach when the hardware buffer capacity overflows or when the architecture is otherwise unable to complete the transaction. This solution can keep the overall high performance of hardware implementations while removing the strict resource restrictions that are inherent in their design (KUMAR et al., 2006; BAUGH; NEELAKANTAM; ZILLES, 2008; DALESSANDRO et al., 2011). Other works aim at improving particular aspects of a Transactional Memory system, applying more advanced algorithms and pieces of hardware that reduce the overhead of transactional execution.

The general essence of Transactional Memory is that it enables the user to specify the minimum of information required (that is, what sections of the code may contend on the access of shared memory data and should be executed as transactions), and the underlying system is allowed to run them as it seems fit, needing only to guarantee that concurrent transactions do not interfere with each other. Compared to more traditional lock-based synchronization semantics, there is a strong expectation that Transactional Memory be overall easier to program and possibly more scalable and more efficient as well.

Although there has been significant advances in Hardware Transactional Memory research, including recent adoptions under the instruction set of mainstream hardware vendors, even when this functionality is present, the method of speculative execution must always rely on Software Transactional Memory for fallback. The software implementation, in its turn, is still overall inefficient, due to the introduction of high overheads from commit/rollback buffer bookkeeping and conflict resolution methods.

1.1 Proposal

While Transactional Memory research promises an easier-to-use method of synchronization over lock-based mechanisms, its adoption strongly depends on whether the execution can be comparably as efficient as locks in the occurrence of real-world scenarios. Therefore, we see then need to approach applications that rely on Transactional Memory systems the same way one would approach more traditional lock-based applications, using automated tools to help humans develop, optimize and debug parallel code. We are not aware of any published memory model that is able to account for the execution of Transactional Memory programs and be used as part of an automated application performance tuner. Our work fits that demand by filling this gap in current TM research.

In this work, we tackle the problem of efficiency in Transactional Memory systems by reducing the execution time of benchmarking algorithms through the profiling and optimization of applications based on a prediction of its runtime behavior. More specifically, we focus on two approaches:

- Extending a Hardware Transactional Memory profiling framework for the collection of a series of parameters that can be employed by an application tuner to automatically select the most efficient Transactional Memory system policies for the application at runtime, by taking into account the expected behavior of the application regarding these parameters.

- Profiling and scheduling of user-level threads on a Software Transactional Memory system based on a prediction of what transactions are expected to conflict, how much rollback overhead will be spent in this potential conflict and what other user-level threads could execute in its place to avoid this overhead and guarantee progress of execution.

1.2 Organization

This remainder of this document is structured in the following way:

Chapter 2 presents the main definitions behind concurrency, synchronization, and speculative execution, in particular Transactional Memory. This chapter provides the conceptual foundation over which the rest of this thesis is structured.

Chapter 3 is an overview of the state-of-the-art regarding hardware-based transactional architectures and simulators, as well as the related work for our Hardware Transactional Memory proposal. Chapter 4 exposes the state-of-the-art for software-based implementations of Transactional Memory, including Software Transactional Memory systems, the LUTS scheduler and related work for our Software Transactional Memory proposal.

Chapter 5 introduces our two proposals for profiling and optimization of Transactional Memory systems, from their conceptual definitions to their implementation. Chapter 6 presents the evaluation methodology and experimental results derived from our work.

Finally, Chapter 7 presents some discussion on the results we achieved, as well as the possibilities of future work and some other final considerations.

2 SCIENTIFIC CONTEXT

Transactional Memory is a parallel programming abstraction that has been researched as a way of expressing data access synchronization. Its main goal is to provide the efficiency of fine-grained locking while eschewing its associated programming complexity (LARUS; RAJWAR, 2006; CASCAVAL et al., 2008).

The transactional approach to parallel programming can be seen as an alternative to the method of locking, wherein multiple threads of execution coordinate their behavior through the notion of mutual exclusion, which guarantees that only one thread will have access to a given memory address at a given time. Section 2.1 presents an overview of this and related solutions to the problem of concurrency control.

At the heart of the Transactional Memory paradigm, there is the concept of a *transaction*. A transaction, in this context, consists of a sequence of operations that is seen as indivisible by any other thread of execution. Its execution is speculative, meaning that it will try to execute completely and commit its modifications, but it is allowed to fail and roll the values modified in memory back to their previous state. Section 2.2 exposes the main definitions and concepts that are necessary to the understanding of the internals of such speculative execution systems.

Many notions from Transactional Memory are analogous to what is required to design Thread-Level Speculation, given that both parallel programming solutions rely on speculative execution, conflict detection and the ability to rollback to a correct prior state of execution. Thus, any description of Transactional Memory would be incomplete without a brief review of the main concepts in Thread-Level Speculation, along with a presentation of its state-of-the-art, which we do on Section 2.3.

While the paradigm behind Transactional Memory presents a fairly straightforward programming interface for transaction delimitation with well-defined semantics, it still allows the underlying system to choose its parameters in a variety of dimensions, such as the interplay between what is implemented in hardware and what is relegated to software, the methods of conflict detection among transactions, and the form of resolution that is employed for transaction conflicts. Section 2.4 presents an overview of these main concepts behind what is currently being researched in Transactional Memory.

2.1 Concurrency Control

The field of parallel programming aims at improving over sequential execution through the division of tasks so that parts of it can be performed by multiple units of execution at the same time. While there exist some problems whose sequential algorithms can be easily transformed into a parallel equivalent, there are those who can only be parallelized as an intertwined set of lines of execution that explicitly cooperate and share partial data

in a coordinated manner.

Given the requirement for this latter pattern of coordinated execution, two methods of communication can be used to explicitly synchronize different lines of execution: message passing and shared memory communication. In the former method, each unit of execution must explicitly send and receive all data that is to be shared through an underlying system of *messages*; in the latter, data is already naturally shared through the main memory, and the units of execution as therefore tasked with the identification of *critical sections* — contiguous areas of code which compete with each other for the access to memory addresses, requiring some method of synchronization between the units of execution to prevent the corruption of data.

While the message passing paradigm is often seen as simpler to use and less prone to obscure programming errors, due to the explicit use of send/receive primitives, it is the shared memory paradigm that is most likely to be used when efficiency concerns are of utmost importance. Given the centralized role that the main memory plays on multicore systems, and the fact that message passing can easily be implemented on top of a shared memory system (but not the other way around), low-level primitives for inter-thread synchronization are usually focused on the protection of shared memory critical sections.

Two of the most widely implemented methods for employing inter-processor data synchronization are the Compare-And-Swap (CAS) and the Load-Linked/Store-Conditional (LL/SC). The former can be seen as an atomic instruction CAS VAL,NEW,MEM, which compares the value stored at location MEM, and, if if is different from the value VAL, assigns into MEM the value of NEW. The latter method, on the other hand, is divided into two different instructions: LL MEM, which returns the value of a given memory address; and SC MEM,VAL, which stores the given value at the memory location as long as the location has not been updated since the execution of LL.

On the software level, these synchronization primitives can be used in the implementation of abstractions known as *locks*. A lock is a piece of memory which stores a value that identifies whether the it has been acquired by any given thread. Any attempt at acquiring a lock that is already being used by another thread will block the flow of execution until the lock has been released.

In the locking paradigm, locking operations must used at the endpoints of critical sections, to guarantee their exclusive access to the data. The identification of critical sections is impacted by the concerns of *correctness*, which requires the encompassing of all conflicting operations under the protection of the lock; *liveness*, which demands that the parallel code be executed without the occurrence of deadlocks; and *performance*, which favors a reduction of critical section sizes, splitting the problem into finer-grained areas of exclusive access to encourage a higher degree of concurrency.

Finer-grained code, favored due to performance considerations, can be notoriously more difficult to be properly programmed and maintained, and goes directly against the correctness concerns which demands that all operations be shielded from concurrent access through the acquisition of an exclusive-access lock during their whole execution.

Aside from these straightforward implementation and performance concerns commonly associated with locks, there are other more subtle drawbacks that emerge from the interaction between concurrent execution of lock-based programs. When a thread holding a contended lock is switched out, all the other threads that are waiting on this lock and are switched in will be forced to yield without progressing, creating a problem known as lock *convoying*. An even worse variant of this problem occurs when a thread

holding a lock is preempted by a higher-priority thread that tries to acquire the lock: the higher-priority thread will have to yield execution and wait for the lower-priority thread to be rescheduled, finish its critical section and release the lock. If a medium-priority thread is available to be scheduled, it may prevent the lower-priority thread from releasing its lock, thus keeping the high-priority thread waiting for the medium-priority one, in a phenomenon called *priority inversion*.

The many shortcomings of the locking paradigm served as motivation for the development of *non-blocking* synchronization algorithms. These algorithms are characterized by their progress guarantees, eschewing the need to *wait* on a lock and relying instead on other methods of accessing contended data. Traditionally, the preferred method of accessing this data has been the use of atomic operations such as CAS and LL/SC.

More recently, there has been a growing interest in the concept of non-blocking speculative execution, wherein concurrent code can be optimistically executed as if no data contention were possible, and any sign of data access conflict is dealt with through the undoing and re-execution of the concurrent code. As long as the access conflict is detected as soon as it occurs, one of the conflicting threads may still be allowed to continue its execution, while its concurrents are signaled to abort and rollback. Except for speculative bookkeeping overhead, it is possible to guarantee that at least one transaction will finish and commit for each actual critical section conflict, allowing speculative implementations to be as efficient as lock-based exclusive-access synchronization in the worst-case scenario, while outperforming locks in the non-conflicting case due to the parallel executions.

2.2 Speculative Execution

Speculative execution is a method of parallel code execution in which critical sections are replaced by *transactions*: sections of the code that access contended data, and are intended to be optimistically executed. The term comes from its homonym in the field of Databases, wherein the optimistic execution consists of tentatively applying all operations, checking whether any conflicts occurred with concurrent transactions, and then either commit or abort/restart the transaction, depending on the occurrence of conflicts.

In database systems, there is a set of required properties that, together, guarantee the proper semantics of speculative execution. These characteristics are known as ACID, from the name of its four components (Atomicity, Consistency, Isolation and Durability):

- Atomicity[1]: transactions must either execute completely, until the end, or fail and rollback, leaving everything as if it hadn't executed.

- Consistency: transactions must take the system from one consistent state to another, both meeting all validation rules.

- Isolation: transactions execute entirely without the interference from other transactions; intermediate states are not seen from other transactions.

- Durability: transactions persist after being committed.

[1]The use of the term *atomicity* differs from the Database community to the Parallel Programming community. In parallel programming, *atomicity* is closer to the database concept of *isolation*, and that is the use we adopt in this thesis, unless explicitly noted otherwise.

In speculative execution systems, the concept of Durability is not usually applicable. The other concepts, however are still required for the execution of the transactions, defining together the proper semantics for the execution.

In order to guarantee the transactional properties, these systems must keep track of the *address sets* employed at each transaction. For a given transaction X, its *write-set* (denoted by X_W) is the set of memory addresses to which X has written during a given execution. Similarly, its *read-set* (written as X_R) is the set of memory addresses from which X has read during a given execution. These sets describe the approximate behavior of a transaction and can be used to estimate the occurrences of conflicts between two concurrent transactions (BOBBA et al., 2007).

When an unresolvable conflict is detected between two concurrently executing transactions, they must be submitted to *serialization*, that is, one of them must be re-executed in a way that precludes conflicting transactions (possibly under an internal lock). For systems where more than one *serialization strategy* is possible, it is a current research problem to define what is the best strategy for a given set of concurrent transactions — that is, when is it better to give up optimistic execution of a transaction and hold re-execute it in a guaranteed thread-safe manner.

Sometimes, a transaction may intrinsically require serialization — for example, if it executes an irrevocable operation like input/output and cannot afford to be invalidated by another concurrent transaction. In the other cases, the underlying system can execute them optimistically and just check whether there has been any memory access conflict. When there is such a conflict between two transactions, the system is required to apply a conflict resolution strategy to decide which transactions must be re-executed.

One way of predicting the likelihood with which two concurrent transactions will need to be serialized is through the analysis of their *conflict rate*, which can be defined for a transaction X as the number of times X has been involved in a data access conflict over the total number of times X executed. We consider that a transaction X is involved in such a conflict if and only if there exists a concurrently executing transaction Y such that $(X_W \cap Y_W) \cup (X_W \cap Y_R) \cup (X_R \cap Y_W) \neq \emptyset$.

Since the occurrence of conflicts demands the rollback of optimistic execution for a further retrial, all speculative execution systems are required to identify and deal with conflicts. This identification must follow a *conflict detection* policy, which specifies at what point in the execution of a transaction the system will check for conflicts. There are three valid policies (HARRIS; LARUS; RAJWAR, 2010):

- detection on open: when a transaction signals to the speculative system before obtaining access to each memory address;
- detection on validation: when a transaction checks during its execution whether its objects have not gotten involved in conflicts since it started running; and
- detection on commit: when a transaction that is keeping all modifications in a temporary buffer decides to write all information to memory and needs to check whether it has not been invalidated by any other concurrent transaction.

Speculative execution of transactions is mainly divided into two different areas of research: Transactional Memory, wherein user code explicitly identifies the regions of code that should be treated as transactions and executed speculatively; and Thread-Level Speculation, which consists of automatically identifying blocks of sequential code that

could be executed in parallel and transforming them into transactions, achieving a type of automatic parallelization.

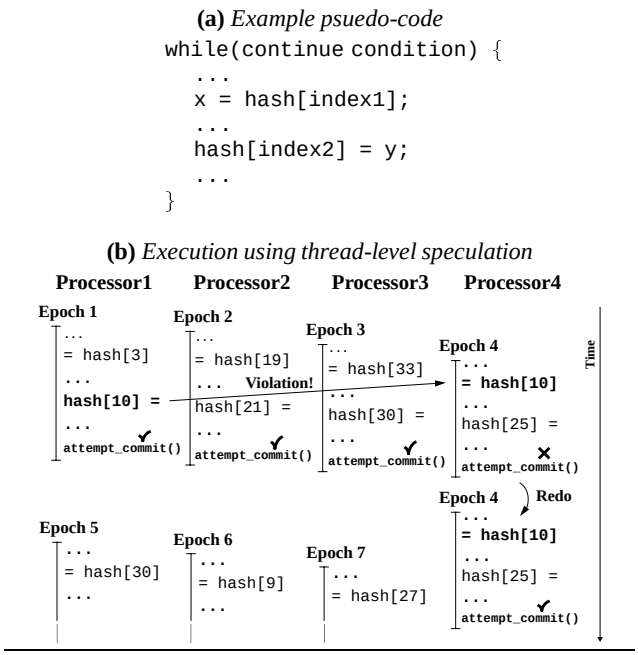
2.3 Thread-Level Speculation

Thread-Level Speculation (TLS) is a method of parallelizing sequential code without explicit semantic input from the programmer (STEFFAN et al., 2005). In this mode of speculative execution, sequential programs are divided into parallel tasks called *epochs*, which correspond to sections of sequential code that the compiler deems suitable to be executed concurrently. We say an A is logically-earlier than epoch B if and only if A would have preceded B in the sequential execution.

This technique differs from more traditional forms of automatic parallelization in that instead of having to prove data independence between sections of code, the compiler optimistically executes them concurrently, assuming that these different epochs executing at the same time are unlikely to run into conflicts. When conflicts do occur, the thread that is executing the logically-later epoch restarts its execution from the beginning. In this scenario, Thread-Level Speculation can be seen as a compiler-generated loop-unrolling into transactions that execute in parallel (GUO et al., 2008) — albeit normally using a more specific speculative execution system for the sake of efficiency.

Figure 2.1 demonstrates an execution where each epoch corresponds to an iteration of a loop. On each iteration, one address of a hash table is read from and another one is written to. In this example, up to four epochs are allowed to execute concurrently. A read-after-write dependency violation between epochs *1* and *4* is detected during the execution of epoch *4*, forcing it to rollback and re-execute.

Figure 2.1: Example of Thread-Level Speculation.



Source: STEFFAN et al. (2000).

During the commit phase of an epoch, the speculative runtime system checks whether there has been any violation of sequential semantics by logically-earlier epochs. If there

has been none, the epoch is free to commit its speculative data modifications and its thread can continue executing. Otherwise, the epoch is *squashed*: it rolls back and its execution is restarted from the beginning.

In order to be able to achieve the proper speculative semantics, a TLS system is required to manage speculative memory state for all threads of execution (GARZARáN et al., 2003). This is akin to version management in Transactional Memory systems. The most commonly used approaches are

- Buffering speculative state in caches or dedicated hardware write buffers, writing data to memory during the commit stage (GOPAL et al., 1998);
- Optimistically writing data directly to memory while generating a log of each memory modification, which is used to roll back the execution when a violation is detected (GARZARáN et al., 2001).

In the second case, in order to be able to roll back memory-altering instructions, Thread-Level Speculation implementations usually rely on special hardware support for the dynamic storage of data indicating what operations have been speculatively executed in memory (GARZARáN et al., 2003).

Thread-Level Speculation can be implemented either completely in software, completely in hardware, or in a hybrid fashion. Software-only TLS incurs too much overhead and suffers from scalability problems, making it impractical (STEFFAN et al., 2005; OANCEA; MYCROFT; HARRIS, 2009). On the other hand, hardware-only proposal are limited by the fact that the hardware needs to decide how to break the program into speculative epochs given only restricted local information (STEFFAN et al., 2005), without the whole knowledge to which a compiler would have access.

Hybrid TLS solutions have been researched where the commit system is implemented in software, while hardware Transactional Memory primitives are used in order to handle the speculative code, relying on the semantics of transactions to deal with the commit/rollback behavior of epochs (GUO et al., 2008; YOO; LEE, 2008a).

Thread-Level Speculation is a promising optimization technique due to the fact that it lifts from the compiler the obligation of proving complete data independency before parallelizing a loop in the source code (STEFFAN et al., 2005). The compiler is thus able to determine what loops should be parallelized based on statically ambiguous dependencies, eliminating the need for precise data dependence analysis and facilitating the parallelization of non-numeric applications, at the smaller cost of having to partition the code into speculative threads of execution (BHOWMIK; FRANKLIN, 2002).

As a method of speculative execution, Thread-Level Speculation can be simpler to use than Transactional Memory, which requires the explicit delimitation of transactions, but it is only applicable insofar as the user code is comprised of a reasonably large amount of loops with few dependencies among its iterations. In the cases that it does succeed, the compiler must still be relied on choose what sections of code to parallelize.

2.3.1 Compiler Support

In spite of having a more relaxed set of constraints than non-speculative methods of automatic parallelization, Thread-Level Speculation still relies on its compiler to decide what loops are likely to have few dependencies among its iterations, and when they should be executed speculatively. The following publicly available open-source TLS compilers have been described in the literature:

- The POSH compiler infrastructure (LIU et al., 2006) extends *gcc 3.5* through the addition of TLS passes that operate on the high-level Static Single Assignment (CYTRON et al., 1991) intermediate representation, partitioning the program into tasks from loops and subroutines;
- The Intel ORC compiler for Itanium architectures (WU et al., 2005), modifying *Open64* to support TLS on IA-64 architectures;
- Stanford's SUIF compiler has been adapted to handle TLS directives by (STEFFAN et al., 2005).
- Purdue University's Cetus (DAVE et al., 2009) compiler infrastructure, which translates input C code into parallel code that uses OpenMP directives.

2.3.2 Benchmarks

Most of the research regarding Thread-Level Speculation relies on executing parallelized sequential code from SPEC INT (WARG; STENSTROM, 2008; ZHAI et al., 2008; ALDEA; LLANOS; GONZÁLEZ-ESCRIBANO, 2011), a heterogeneous set of applications often used in performance measurements of optimized sequential applications.

Traditionally used parallel benchmarks like NAS (BAILEY et al., 1991), SPLASH-2 (WOO et al., 1995), along with more recent benchmark sets like PARSEC (BIENIA et al., 2008) are also commonly used in the TLS research, as a way to assess how good a TLS implementation is when compared to hand-coded parallelization, but the already inherently parallel nature of these benchmarks prevents the extraction of any heavy conclusions from the comparison.

2.4 Transactional Memory

Parallel programming under the Transactional Memory (TM) paradigm constitutes mainly in specifying the proper delimitation for all transactions in the source code. The execution of these transactions under the right semantics is a requirement that must be satisfied by any implementation of a TM system. Nevertheless, there are several internal aspects to the implementation of such a system that are not strictly defined by the speculative semantics, and those aspects still comprise an active area of research.

One of the main dimensions through which different designs of Transactional Memory can distinguished themselves is that of whether they are implemented directly on hardware, or using low-level library primitives. While Hardware TM implementations are physically limited and will fail to execute transactions that are too long or that execute some unsupported operation (DICE et al., 2009), current Software TM implementations can still be too slow to be used in high-performance applications (CASCAVAL et al., 2008). Notwithstanding current research on both hardware and software fronts, hybrid implementations have also been proposed as viable solutions, providing an interface that is similar to the one used for software systems, and using the underlying hardware resources available as long as possible, with the guarantee of falling back to the software implementation on failure (KUMAR et al., 2006).

Some implementation concerns are pertinent to the design of every Transactional Memory system. For example, all such systems are required to somehow detect conflict among concurrent transactions, and to be able to apply some method of conflict resolution, which is tasked with the decision of which transactions to rollback in the face of

a detected access conflict. The method and the timing of *conflict detection*, as well as the form of *conflict resolution* employed will directly influence the performance of the execution. In fact, these things may be somewhat merged into a single entity under some systems, which use past knowledge of detection and resolution to decide how to handle further conflicts.

Conflicts between two transactions may also be derived from a limitation of the underlying transactional system. Some hardware-based systems will store transaction data and identification in an extension of one of the cache memories, where the mapping of different addresses into the same cache line may cause spurious conflicts to arise, either in the case of both addresses being close enough to be part of the same line, or due to both addresses being mapped into the same line in spite of being completely unrelated.

This latter cause of conflicts, occurring between unrelated addresses, may afflict software systems as well, since they store transaction identification in a hash table indexed by the accessed address — and it would generally be impractical to dedicate half of the available main memory just to guarantee that every address accessed in a transaction could be handled by the Transactional Memory system. The characteristics of this hash table, such as the hashing function, the hash table size, or granularity of the conflict between two addresses, are still an area of research, since the appropriate values may vary between programs.

Another design dimension of a Transactional Memory system is its *version management*, that is, the mechanism through which transactional TXWRITE operations are propagated. In an eager version management, writing operations are executed directly in a shared memory, forcing the conflict detection to also behave eagerly, and demanding a rollback operation that explicitly undoes those writes. On the other hand, when version management is performed lazily, each transaction will use some local memory to store transactional data, relegating the task of updating the shared memory to the TXCOMMIT operation.

Even in the case of lazy version management, the underlying system is still required to use a scheme of synchronization at some point in the execution of the transaction, whether it be through special cache-memory targeted instructions, locks or non-blocking algorithms. This existence of such a synchronization point in the semantics of transactional programs establishes that any ordinary transaction may be subject of abortion and rollback.

In some cases, the possibility of failure is not acceptable for a given transaction, due to the presence of an *irrevocable* operation — such as a system call or an input/output operation, which produces side-effects that cannot be rolled back. Under such circumstances, the speculative system must be able to execute the transaction non-speculatively, preventing any conflicting transactions from committing as long as the irrevocable transaction itself hasn't committed. Compiler preprocessing may be able to identify some of these irrevocable transactions beforehand, but in the general case, the detection of such transactions must still be performed at the first attempt at an irrevocable operation.

2.4.1 Compiler Support

Compiler support for hardware systems consists of generating the appropriate calls to hardware instructions that implement transactions. Hybrid or software-only support, on the other hand, requires that appropriate calls to transaction controlling primitives be generated, transferring the flow of execution to a runtime library before and after the execution of a transaction.

Compiler support is not a strictly necessary tool for Transactional Memory, given that it requires no specific compile-time preprocessing; in some cases, even the sole delimitation of transactions with the appropriate TXBEGIN/TXEND instructions or library calls will suffice. However, when present, it allows the target programming language to hide the implementation details of the transactions through a syntactically unified interface. Moreover, it permits the preprocessing of global information that can be used in code optimization during the execution.

One of the works in this area is an extension to the Intel ICC compiler that has been used to investigate code optimization techniques for Transactional Memory constructs in the C language, generating code for software implementations (WANG et al., 2007). Taking a different approach, the OpenTM API (BAEK et al., 2007) specifies an extension to OpenMP (CHAPMAN; JOST; PAS, 2007) where transaction-related keywords indicate how the hardware instructions or runtime functions should be called during the execution. This API has been implemented over the GNU OpenMP environment.

The GCC compiler collection has defined a full range of Transactional Memory operations for their C compiler, including syntactic sugar for transactions themselves, as can be seen in Figure 2.2.

Figure 2.2: Example of a list-traversing transaction on GCC.

```
__transaction_atomic {
   next = set->head->next;
   while (next->val < new_val) {
      next = prev->next;
   }
   next->val = new_val;
}
```

The presence of a compiler also allows for the preprocessing of transactional information, giving way to algorithms that rely on some non-local advanced information. For example, the technique of Selective Reconciliation relies on compile-time computed information regarding data dependency inside a transaction, in order to avoid having to rollback and re-execute the whole transaction, only having to re-execute the conflicting access instead (MANNARSWAMY; GOVINDARAJAN, 2012).

2.4.2 Benchmarks

Transactional Memory benchmarks focus on realistic examples that are not trivially parallelizable in a lock-free manner. The goal is to compare the performance of these implementations against their lock-based and sequential counterparts under varying workloads. The benchmarks that are most often referenced in the TM literature are:

- The STAMP benchmark suite (MINH et al., 2008) comprises a set of eight non-trivial applications, with a sequential and a transactional version for each application, along with corresponding input workloads. Table 2.1 presents a summary of the operations that the STAMP benchmark executes.

- The STMBench7 benchmark (GUERRAOUI; KAPALKA; VITEK, 2007) provides a fine-grained and a coarse-grained lock-based implementation of a database graph-traversal application, which can be compared for performance with a TM version of the algorithm. Different workloads can be selected, with varying amounts of data contention.

Table 2.1: Summary of STAMP benchmarks.

Benchmark	Description
bayes	Learning the structure of a Bayesian network through a hill-climbing strategy using an ADTree data structure.
genome	A gene sequencing program, which reconstructs a gene sequence given some segments from the whole gene.
intruder	A network intrusion detection algorithm, which scans network packets looking for matches to intrusion signatures.
kmeans	Applying the k-means clustering method to partition a set of observations into a given number of clusters based on their proximity.
labyrinth	Finding the shortest path between two points in a maze.
ssca2	Graph construction represented with adjacency arrays.
vacation	Simulator of a travel reservation system, where all threads try to concurrently interact with a single database.
yada	An implementation of the Delaunay refinement algorithm, which creates Delaunay triangulations for a given set of points.

- The Lee-TM benchmark (ANSARI et al., 2008) amounts to a circuit routing algorithm whose implementation requires various transaction lengths. Both a sequential version and a transactional version are implemented.

Given that the Transactional Memory syntax still has not been standardized under different compilers, and some compilers have not even been developed for this syntactic support, instead of using any particular syntax, the C language benchmarks found in the literature use macro calls from the C preprocessor to identify transactions in the code.

2.4.3 Lock Elision

An interesting consequence of the development of speculative execution systems is that legacy lock-based programs can be speculatively executed under a technique known as Lock Elision, without any extra efforts on the part of the programmers. When a lock-based code is executed under Lock Elision, its critical sections are treated as if they were transactions, and writes to the critical section's lock are not performed during the transaction.

As a fallback mechanism, if an execution under Lock Elision conflicts with a concurrent critical section, a lock can be acquired to revert back to the old lock-based paradigm. This ability to execute code speculatively without requiring major software rewriting is what prompts many software and hardware approaches to Transactional Memory to provide support for Lock Elision alongside the explicitly speculative operations. Deciding the best time to abandon Lock Elision in favor of standard locking for a given transaction is still an open research question.

2.4.4 Basic Research

Research often takes advantage of low-level hardware primitives to provide raw efficiency, or sophisticated software algorithms that guarantee flexibility on the behavior of the speculative system. Chapter 3 in this thesis will focus on hardware approaches, while Chapter 4 will present the software-based research.

Research on hybrid Transactional Memory usually relies on using hardware primitives

to implement the common-case scenario and fallback on a software-based implementation when appropriate. Since the programming interface is usually the same as in software Transaction Memory, research usually focuses on the hardware aspect of the hybrid implementation (BAUGH; NEELAKANTAM; ZILLES, 2008) and on deciding when to use a hardware-only implementation and when to move to a software-based solution (KUMAR et al., 2006).

Some works aim at the development of conflict resolution strategies that ensure some formal guarantee of forward progress for conflicting transactions (SPEAR et al., 2009). Other works are directed at identifying techniques to manage specific commonly occurring types of data conflicts (WALIULLAH; STENSTROM, 2014). Rather than specially handling conflicts after their occurrence, a further solution involves the *scheduling* of transactions based on their previous execution, aiming at avoiding the conflict of transactions in the first place (NICÁCIO; BALDASSIN; ARAUJO, 2011).

In spite of the clear semantics proposed for transactions in relation to concurrently executing transactions, their behavior under nesting is a non-trivial topic that is still being researched (PERI; VIDYASANKAR, 2013). This interaction between inner and outer transaction can be generally divided into three possibilities of *nested transactions* (HARRIS; LARUS; RAJWAR, 2010):

- Flattened nesting: Closely follows the semantics of reentrant locks in the locking paradigm, wherein the nesting itself is all but ignored. Committing the inner transaction has no visible effect. Aborting the inner transaction aborts the outer one as well, retrying the whole transaction.

- Closed nesting: Committing the inner transaction has no visible effect. Aborting the inner transaction rolls it back and retries its execution alone.

- Open nesting: Committing the inner transaction propagates its changes to other transactions immediately, and even if the outer transaction is later aborted, the changes are not rolled back. Aborting the inner transaction rolls it back and retries its execution alone.

Transactional semantics is also a topic of research under operating systems. One of the long-standing issues of POSIX operations is the fact that, while they do provide the necessary atomicity for individual system calls, there is no way to group those calls into a larger atomic unit. TxOS is a modification of the Linux kernel that provides such support for transactional access of operating system resources: programmers can delineate the system-level transactions with the system-call functions `sys_begin` and `sys_end`, aborting all intermediary system calls with `sys_abort`, if necessary. TxOS can enable the speculative execution of several otherwise irrevocable transactions; however, some system calls, such as the ones that perform network operations, may still require execution under a non-speculative mechanism (PORTER et al., 2009)

Under the current concerns of energy efficiency, it is no surprise that some research be dedicated towards a more energy-efficient implementation of Transactional Memories, especially considering the explicit waste associated with aborted operations which must be rolled back and re-executed. By designing and deploying a special commit protocol on dedicated hardware, both execution time and energy consumption may be reduced (GAONA et al., 2013). The accommodation of higher-level techniques obtained through theoretical work into hardware implementations of Transactional Memory is

also responsible for some reasonable gains in energy efficiency (WALIULLAH; STENSTROM, 2014).

Finally, on the topic of more theoretical works, some effort has been put on the task of analyzing the general properties of Transactional Memory programs. For example, by investigating how serialization influences the throughput and efficiency of Transactional Memory systems based on the workload, the impact of conflict resolution techniques can be more readily evaluated (HEBER; HENDLER; SUISSA, 2012). By axiomatizing the partial ordering of transactions, very precise definitions can be achieved regarding operations such as *serialization*, specifying correct behavior and allowing for the formal verification of Transactional Memory systems (DOHERTY et al., 2013).

3 HARDWARE TRANSACTIONAL MEMORY

The ideas behind Transactional Memory were first proposed by (KNIGHT, 1986) as part of a method for automatically parallelizing sequential code, which would execute speculatively in a special hardware designed to use cache coherency as the means to maintaining correctness of the parallelized code, somewhat similar to the current proposal of Thread-Level Speculation.

Similar concepts were later used by (HERLIHY; MOSS, 1993) when defining a new processor architecture that strove to make lock-free synchronization as fast as the more traditional lock-based synchronization solutions. The central goal here was to allow programmers to define custom memory access code that spanned over multiple independent memory addresses while keeping the property of atomicity found in the more restricted single-address read-modify-write methods.

This architecture was called Transactional Memory, and it defined the main operations that comprise a modern Transactional Memory system: TXBEGIN, TXREAD, TXWRITE, and TXCOMMIT. The authors proposed the addition of an extra auxiliary cache to store transactional data, and specified some custom changes to standard cache coherence protocols that allowed for the implementation of the defined instruction set. In particular, they described how to adapt the Snoopy protocol (GOODMAN, 1983), augmenting it with tags that describe the transactional status of a given piece of data.

Since the aforementioned proposals for Transactional Memory, much research has been done towards an efficient implementation that could be introduced into a mainstream processor. In this chapter, we present an overview of the state-of-the-art regarding those hardware implementations. Section 3.1 looks at the hardware implementations that have been put forth by the industry. Section 3.2 presents further hardware designs, in the form of current hardware simulators that support Transactional Memory. Section 3.3 analyzes Syncchar, a tool which combines a hardware simulator with a transactional memory system and a performance tuning mechanism.

3.1 Hardware Proposals

In the last few years there has been substantial progress towards a more widely-accessible hardware implementation of Transactional Memory. Most research concerns implementations that extend currently existing hardware architectures with transactional operations and special hardware to handle the transactions. These solutions offer a best-effort hardware TM support, providing instructions that can be used to delimit hardware transactions, but that are allowed to fail.

We present the Transactional Memory designs that have been, or are currently being researched by the hardware industry and have been discussed in the literature.

3.1.1 Vega

The first commercial system that fully supported the semantics of Transactional Memory in special-purpose hardware was the Vega machine, by Azul Systems (CHOQUETTE; TENE; NORMOYLE, 2008). This architecture features up to 16 processors, each with 54 cores, running concurrent Java applications on its 64-bit RISC processors. The underlying scenario is one in which costumers wish to write standard lock-based multithreaded Java programs and be able to use all 864 cores efficiently.

As a solution to the lock efficiency requirement, Azul decided to build Transactional Memory support directly into the Java Virtual Machine, in order to accelerate lock-protected operations without any modifications to the costumer's code. The Virtual Machine makes use of three transactional instructions: SPECULATE, which enables speculative execution for loads and stores; ABORT, which interrupts speculation mode and rolls back to the beginning of the transaction; and COMMIT, which finishes the speculation mode, either flushing the written data to main memory or aborting execution and rolling back to try again (CLICK, 2009).

The Java Virtual Machine is implemented in such a way that, whenever it sees a lock, it decides whether to actually acquire the lock or to try to speculate that section of code in a transaction, effective executing a conditional Lock Elision. Initially, it tries to speculate, but if the speculative execution fails too often, it switches back to acquiring the given lock in the first trial.

The hardware organization for the Vega machine is one where each core is connected to a private L1 cache and to a L2 cache that it shares with other 8 cores. Support for Transactional Memory is implemented through the addition of two extra bits on each L1 cache line: *speculatively-read* and *speculatively-written*. Due to the fact that no changes were performed on L2, cache misses on L1 will always abort an executing transaction.

Early experiments showed that memory access conflicts would restrict speedup over sequential execution to less than 1.1. Hardware capacity overflow, on the other hand, was reported to be very uncommon, and did not impact the runtime efficiency.

3.1.2 Rock

Sun Microsystem's approach to speculative execution was the implementation of a *checkpoint*-based architecture on their Rock processor, in which a checkpoint of the current hardware state could be taken at an arbitrary instruction. This architecture was conceived as part of a scheme for improving the efficiency of executable code through the execution of the same software thread in two different physical threads. Whenever the first thread became blocked due to a high-latency operation (such as an L1 cache miss or a slow floating-point operation), the other one could speculatively execute the next instructions, and, if no conflict was detected, those operations could be committed (CHAUDHRY et al., 2009).

The checkpoint-based design of Rock allowed it to easily support Hardware Transactional Memory. The instruction CHECKPOINT <FAIL_PC> can be used to explicitly define a rollback point, copy the current value of registers and start executing in the speculative mode; if this execution fails, the registers can be rolled back to their values at checkpoint and execution proceeds at FAIL_PC. Alongside the checkpointing behavior of registers, support was also added for some of the memory-accessing operations at 32 of the L2 cache lines. Unsupported instructions would automatically abort the transaction.

Due to the need to version speculative cache lines, Rock's hardware implementation

involved the addition of k bits to each of the L2 cache lines, one for each of the k possible physical threads. Every speculative memory access sets the appropriate bit, indicating that the cache line is currently storing speculative data. Whenever a cache line is invalidated or evicted, the transactions that have accessed it are aborted. During the execution of the COMMIT instruction, cache lines that have been accessed by the current transaction are locked to block access from other threads until the transaction has been properly committed.

The Rock processor was poised to be released at some point in 2009, and early research revealed some encouraging results in terms of speedup (DICE et al., 2009), but the hardware project was canceled before its release, which prevented it from reaching a wider programmer audience.

3.1.3 ASF

In 2009, AMD published the Advanced Synchronization Facility (ASF), a proposed extension to the AMD64 Instruction Set Architecture that defined new instructions to support Transactional Memory semantics (CHUNG et al., 2010). The proposal revolved around the SPECULATE instruction, which sets a bit in the flag register and starts a transaction; and the COMMIT instruction, which either finishes the transaction, or rolls back, jumps to the instruction right after SPECULATE and clears the flag bit. Inside a transaction, LOAD and STORE maintain their non-transactional behavior, while the LOCK MOV operation performs the corresponding memory-to-register or register-to-memory transactional move. Two remaining instructions can be used used to force a transaction to ABORT or to ignore (RELEASE) a memory address when checking for conflicts.

The proposal for the ASF extension was made available to the public as a way of gathering input from the parallel programming community. It was presented as a set of low-level hardware instructions that could be used as the means to implement anything ranging from lock-free data structures to full-fledged Software Transactional Memory systems. Ultimately, the goal was to adapt the design of this extension according to the result from the review of its users. Internal research using simulators deemed the particular semantics of an implementation of ASF to be viable (CHRISTIE et al., 2010), but no further pronunciations have been made from AMD on the implementation of this proposal in actual hardware.

3.1.4 Blue Gene

The first widely-deployed implementation of Transactional Memory semantics to support commonly used systems languages (such as C and C++) was IBM's Blue Gene/Q processor (WANG et al., 2012), developed for the Sequoia supercomputer. In this architecture, speculative state is stored in a multi-versioned 16-way set associative L2 cache, where each speculative version of a cache line is stored in a different L2 way.

The hardware organization for this machine was originally designed without support for Transactional Memory. As a consequence of this, the private L1 caches were not tailor-made for the storage and management of speculative data. Rather, this data is stored in the L2 cache, which is shared among all 16 cores of a Blue Gene/Q processor, and is what restricts the scope of the Transactional Memory to each processor in this supercomputer. The L2 keeps track of which thread owns each speculative version of a cache line, which allows for the detection of conflicts between different concurrent transactions.

Since the speculative execution buffers its state in the L2 cache, this processor implements two possible solutions to provide the correct interaction between L1 and L2, in the

form of two modes of transaction execution. In what is called the short-running mode, any store to an address will evict its line from L1, and subsequent access to this address will miss L1 until the transaction commits. In the long-running mode, TLB aliasing is used to version the address space in L1, allowing it to be used during the transaction, but requiring that the whole L1 cache be invalidated at the beginning of the transaction.

Conflict resolution is performed when the hardware eagerly detects an access conflict and triggers a conflict interrupt. Alternatively, these interrupts can be disabled so that the conflicts may be checked lazily at commit time. In this particular implementation, the kernel handles conflict interrupts by favoring older transactions over transactions that have a more recent starting time.

Forward progress of repeatedly failing transactions is ensured through an *irrevocable mode*, which essentially transforms the transaction into a locked critical section that executes non-speculatively. Only one transaction can execute in irrevocable mode at a time, and transactions that conflict with an irrevocable transaction will always fail their speculative execution.

While the overhead of single-thread execution was lower than the overhead on STM systems, it was still significant, being as low as about 0.5 in the worst-case STAMP benchmarks. On the other side of the spectrum, in the best-case scenarios, a 64-thread execution yielded speedups that closely matched the 16 maximum speedup achievable in this 16-core machine.

3.1.5 Haswell

In early 2012, Intel released the specification for Transactional Synchronization Extensions (TSX) (Intel Corporation, 2012), part of their then-upcoming Haswell architecture, which was officially announced later at 2013, and started shipping at the end of the same year. The proposed instruction set defines the XACQUIRE and XRELEASE instruction prefixes for hardware Lock Elision, which comprises Haswell's main interface for speculative execution. If such a speculative execution aborts, it is resumed at the instruction that was prefixed by XACQUIRE, this time re-executing the code section without Lock Elision. These instruction prefixes are simply ignored in older hardware, which just performs locking under the traditional semantics.

A second part of the new set of instructions deals with explicit transactional semantics through the XBEGIN, XEND and XABORT operations, with no restriction on what instructions can be executed inside a given transaction (Intel Corporation, 2013). The XABORT instruction is required to provide a label to a fallback non-transactional implementation, to be executed in the presence of repeated aborts.

At an organizational level, the hardware keeps track of addresses in the read-set and the write-set at a 64B cache-line granularity. Only the L1 cache can store transactional versions of an address's data, and any eviction of transactional data will cause an abort. In order to manage the transactional data, the 8-way set associative L1 cache can store a speculative version in each of its ways, leaving the cache coherence protocol to detect data conflicts and abort the thread that detected the conflict.

Speculative execution of irrevocable actions, such as system calls and input/output operations, is not supported in any way, and any attempt will abort the executing transaction. These operations can only only succeed under non-elided locks in a non-transactional implementation of the critical section.

Differently from other hardware Transactional Memory implementations, Intel's design is primarily aimed towards increasing the performance of legacy lock-based code

through Lock Elision, with a mere secondary goal of promoting new synchronization strategies through its transactional capabilities. Hence, it is a reasonable expectation that speculative benchmarks perform better than their lock-based counterparts.

Some works have used Haswell to attain high speedup previously unseen in lock-based implementations. For example, Haswell's Transactional Memory has been used in an in-memory database, achieving near-linear speedup for database lookups on up to 8 threads, and about 16 speedup for 32 threads (VIKTOR LEIS ALFONS KEMPER, 2014). In spite of such results, however, no performance results for mainstream Transactional Memory benchmarks have been published to date for the Haswell architecture.

3.2 Simulators

In order to be able to evaluate an architecture that still hasn't been built, it is necessary to use a program that simulates the target hardware. Seeking to understand what simulators tend to be used in the Transactional Memory research, we have scoured the literature looking for papers published during the past 2 years on top-tier conferences, attending to their method of architecture evaluation. Table 3.1 presents a summary of the simulators that were described in the literature.

Table 3.1: Simulator usage in the TM literature.

Simulator	# Papers
GEMS	3
GEMS (ATMTP)	2
GEMS (LogTM)	4
JavaSim	1
M5	1
Mambo	1
MPARM	1
PTLsim (ASF)	2
SESC	1
Simics	7
Simics + MetaTM	2
Simics + GEMS	10
UVSIM	1
Total	36

Table 3.1 was build by looking at the results of a Google Scholar query to *"transactional memory" simulator*. Out of all papers analyzed, thirty-six of them mentioned which simulator was used. While the simulators we evaluated targeted TM specifically, we expect the same simulators to be able to be used in Thread-Level Speculation research.

Most of the research on simulated Transactional Memories uses some extension to the full-system x86 simulation infrastructure known as Simics MAGNUSSON et al. (2002). The Simics-based GEMS simulator MARTIN et al. (2005) is a widely used research tool for Transactional Memory, along with its variants using the modules ATMTP (MOIR; MOORE; NUSSBAUM, 2008), which models the canceled Rock processor (described in Section 3.1.2); and LogTM (MOORE et al., 2006), modeling a homonymous Transactional Memory system that substitutes the snoopy cache-coherency protocol by a fast-committing directory-based alternative.

Other well-known simulators in the Transactional Memory area are the PTLsim-ASF YOURST (2007), simulating AMD's ASF extension for 64-bit x86 architectures (described in Section 3.1.3); and the SESC RENAU et al. (2005) emulator, for MIPS instruction set architectures. Other simulators have all been developed for the specific work performed on their respective research papers, and do not seem to possess the required generality to be adopted elsewhere.

Most simulators center their Transactional Memory support around a snoopy cache-coherency protocol such as XMESI (ROSSBACH, 2011), a modified MESI protocol with five additional states for cache lines that are accessed in a speculative context. Table 3.2 lists the name of these states, with a brief description of the respective cache line state.

Table 3.2: Additional states in XMESI.

State	Description
TS	shared (S) cache line that has performed a TXREAD.
TMU	modified (M) cache line after a TXREAD.
TMM	modified (M) cache line after a TXWRITE.
TQS	cache line invalidated by another's previous TXREAD.
TQM	cache line invalidated by another's TXWRITE.

The additional states augment the standard MESI coherency protocol in such a way that it is able to represent the effect of transactional operations on cache line validity, keeping track of possible data access conflicts. Interaction between transactional and non-transactional code invalidates the cache line that is storing transactional data.

This aforementioned transactional cache-coherency protocol was born as part of a Simics-based Hardware TM system simulator known as MetaTM (RAMADAN et al., 2008). Its goal was to be able to provide the necessary semantics to substitute lock-based critical sections by transactions inside the Linux kernel, in order to analyze the impact of different conflict resolution techniques in a fully transactional environment.

MetaTM is based on an multiprocessor architecture, where each processor is associated to a L1 cache implementing transactional semantics through the XMESI protocol, and a private L2 cache following a standard MESI protocol. All transactional behavior is therefore restricted to the processors and the L1, and evictions from this cache will abort an executing transaction.

Since the transactional data is kept on cache, and the cache coherency protocol is capable of identifying data conflicts at the moment they happen, MetaTM can use this eager conflict detection mechanism to apply any of a variety of conflict resolution policies, as presented in Table 3.3.

Alternatively, instead of resolving conflicts by rolling back competing transactions, MetaTM also implements a mechanism through which a transaction that has just performed a conflicting instruction can undo this operation and stall until the other transaction commits instead, avoiding the overhead of a rollback and re-execution in those cases.

When a conflicting transaction needs to be re-executed, a *backoff policy* is applied to determined how much time the transaction should wait before restarting. This behavior is useful in cases where an immediate restart would most likely repeat the same conflict and abort it again. Table 3.4 presents the backoff policies available in MetaTM.

While other Hardware TM systems will often provide direct support for nested transactions, implicitly handling reentrancy of transactions, MetaTM supports instead the ex-

Table 3.3: Conflict resolution policies in MetaTM.

Policy	Description
Timestamp	Oldest transaction wins.
Kindergarten	Transaction that detects conflict relinquishes its execution and loses, as long as it has not been aborted before. If both transactions have aborted before, Timestamp is used.
Karma	Transaction that has accessed the largest amount of addresses during its executions wins.
Eruption	Similar to Karma, but artificially boosts future address count for losing transaction by the number of addresses accessed by the winner.
SizeMatters	Transactions that has read/written the largest number of bytes during its executions wins.

Table 3.4: Backoff policies in MetaTM.

Policy	Description
None	Use no backoff and restart immediately.
Random	Wait a random amount of CPU cycles (between 0 and 1000).
Linear	Wait a number of cycles that is proportional to the number of times the transaction has already backed off.
Exponential	Wait a number of cycles that is proportional to an exponential on the number of times the transaction has already backed off.

plicit management of a transaction stack. The operation XPUSH can be used to push the identifier for the current transaction onto the stack, leaving the thread in a non-transactional mode; its counterpart, XPOP can resume the transaction from the top of the stack. When an interrupt is executed, the active transaction is suspended through an implicit XPUSH, allowing further transactions to be used even inside interrupt handlers.

The modifications to hardware organization outside from the processors revolve around the conversion of the L1 cache to the XMESI protocol, along with the addition of a per-cache-line field to hold a transaction identifier indicating the owner transaction for of each cache line. Alongside XPUSH and XPOP, the processor also provides the XBEGIN and XEND instructions, which are used to control the scope of transactions.

Irrevocable transactions are supported under a software implementation that is added to the simulated Linux kernel. Operations that normally cannot be rolled back, such as network access, are executed inside a spinlock-based critical section that uses the XCAS instruction: an implementation of CAS that cooperates with the transactional modules of the simulator, triggering a data access conflict on concurrent transactions if necessary (HOFMANN; ROSSBACH; WITCHEL, 2009).

3.3 Syncchar

One of the current obstacles to the employment of Transactional Memory in applications with high efficiency requirements is the lack of knowledge regarding the performance aspects that characterize these applications. In particular, while the tuning of

lock-based code follows a well-known procedure which consists in identifying critical sections that rely on contended locks and modifying their data access patterns to alleviate that contention (possibly also breaking them into finer-grained critical sections), the tuning of transactional code will often consist in identifying the contended memory addresses themselves and diminish transaction dependence on those addresses whenever possible.

Over the last years, some attempts have been made at modeling the behavior of Transactional Memory applications in order to support programmers at the task of optimizing their performance. The Syncchar performance modeling tool is an example of this type of research (PORTER; WITCHEL, 2010). Syncchar instruments code compiled under either a lock-based or a transaction-based paradigm, and executes it in a simulator, in order to identify a model that characterizes the synchronizing behavior in the code and to be able to predict what portions of code should be modified to improve the performance of the application when executed under a transactional paradigm.

Execution of a parallel application Syncchar can be divided according to its three main phases: pre-processing, simulation and post-processing. During the pre-processing phase, Syncchar parses an object dump of the application and identifies the value of the Program Counter register at all points of synchronizing instructions (such as instances of `xcas` for lock-based code, or `load/store` for transactional code). These positions of synchronizing instructions are later turned into memory breakpoints, allowing them to be tracked during the simulation, which generates a log of this synchronization activity. This log can then be post-processed, producing a concise report of memory contention and transactional conflicts.

Syncchar is implemented mainly as a C++ module under the Simics hardware simulator, as an extension of the MetaTM design. Since MetaTM already expands the x86 instruction set with Hardware TM operations (implemented as Simics "magic" instructions), Syncchar only needs to set up the tracking of synchronizing instructions and implement their logging through callback functions.

Aside from this tracking module, Syncchar also relies on a high-level Simics configuration file to specify the parameters of the execution, such as the conflict resolution and backoff policy. A `$benchmarks` variable describes the benchmarks that will be automatically launched when the simulated kernel boots up. Since Simics is able to simulate the whole system stack, only minor modifications to a Linux kernel's boot process should be enough to accommodate the execution of benchmarks.

Due to all of the inherent shortcomings of Hardware TM, there is a strong margin of improvement through the use of tuning that has not yet been widely explored. The Syncchar tool represents one of the first steps in this direction, modeling and predicting the behavior of TM applications, but further work would still required if one wishes to use this information to automatically tune Transactional Memory applications.

4 SOFTWARE TRANSACTIONAL MEMORY

Following the seminal work that popularized the concept of Transactional Memory through a hardware implementation (HERLIHY; MOSS, 1993), another group of researchers published a proposal that relied on the opposite approach: rather than requiring a whole redesign of the standard processor organization and architecture, their Software Transactional Memory was able to support the expected semantics on conventional hardware, requiring only the commonly-found LL/SC instruction (SHAVIT; TOUITOU, 1995).

This proposal required that each word of transactionally accessible memory be paired with another word of memory to store a pointer to its *owner* transaction. Programs were then required to provide, during the TXBEGIN operation, a list with all memory addresses that could be accessed inside the current transaction. The transactional semantics were a result of atomically setting the owner for all of those memory addresses before the execution of the transaction, and the subsequent ownership release during TXCOMMIT. Since the ownership semantics was achieved at an exclusive-access basis during TXBEGIN, this design followed closely the existing lock-based paradigm.

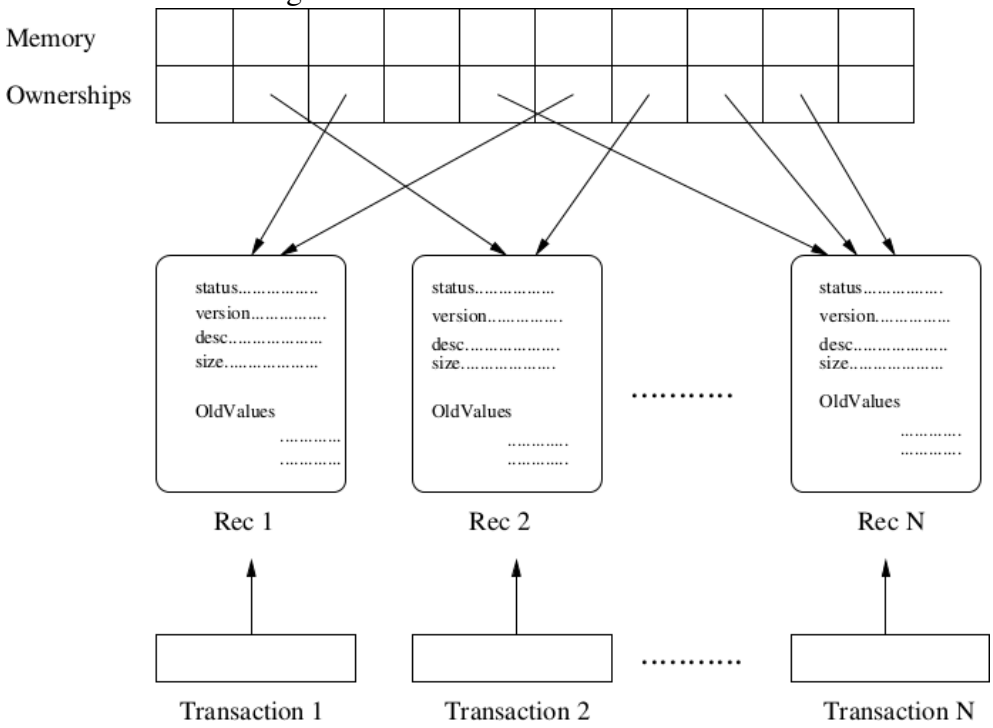
Figure 4.1 presents the general schematics of this Transactional Memory implementation. Each transaction is internally represented by a Record, which stores information regarding its current status, size, and the owned values it has modified. Note that their system is built upon a one-to-one correspondence between memory addresses and ownership pointers, which effectively requires that half of the available main memory be dedicated towards bookkeeping.

In this design of Transactional Memory, a transaction must first be assigned as the owner for each of the addresses it accesses before its code can actually execute. If a given address is already owned by another transaction, the current transaction will fail, and will attempt to *help* the conflicting transaction by executing it in the current thread before trying to execute its transaction again.

Due to this helping behavior, the method of transaction synchronization guarantees a non-blocking execution, in which, even if the thread for a conflicting transaction is scheduled out, the transaction itself may still commit regardless of scheduling priority. This avoids the common problems of deadlock, priority inversion and convoying; in order to avoid the problem of livelocks, address ownership must be acquired in an increasing order.

While being a historically important Software Transactional Memory design, this early proposal is severely limited in some aspects. Aside from the hefty requirement of half the available memory for bookkeeping, the TXBEGIN operation also demanded a list of memory addresses that would be accessed inside the transaction — this list would be used to acquire ownership in a well-ordered manner, ensuring that no deadlock would

Figure 4.1: Software TM schematics.

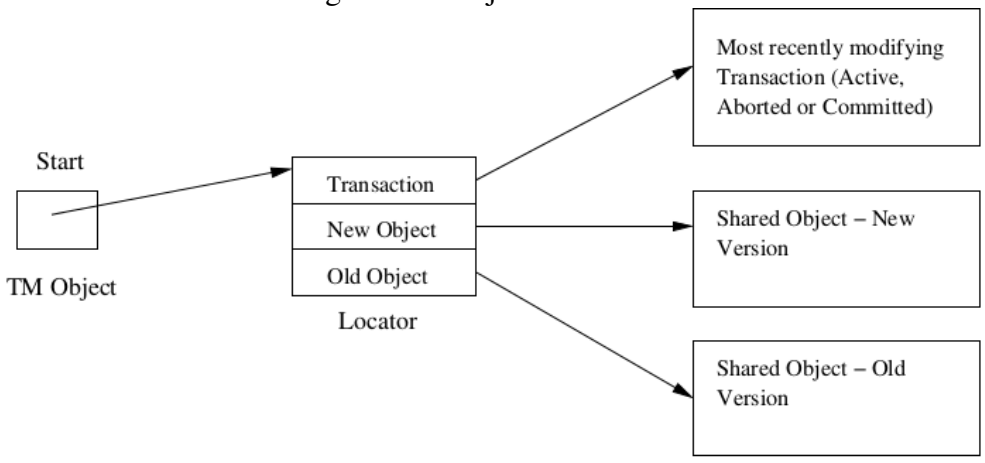


Source: SHAVIT; TOUITOU (1995).

be possible from the execution of two concurrent transactions.

An entirely different approach is to move away from word-based Software TM and use an object-based approach. In this extension to Object-Oriented Programming, each object pointer is replaced by a pointer to a Locator object instead, which itself stores a pointer to the target object, as can be seen in Figure 4.2.

Figure 4.2: Object-based TM.

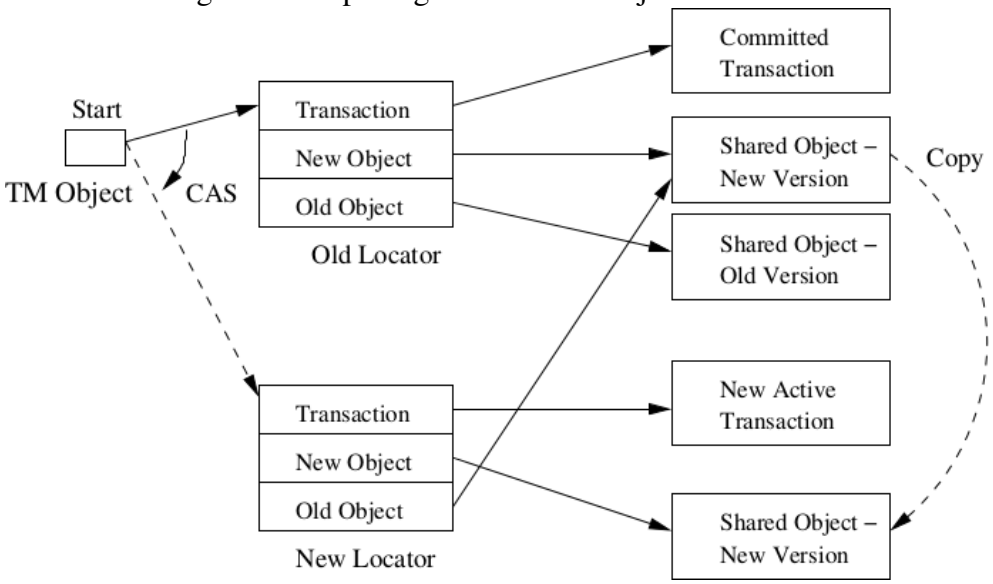


Source: HERLIHY et al. (2003).

The Locator stores a pointer to the transaction that most recently tried to modify the object, along with both the old and the new versions of the object. Whenever a transaction wishes to access an object, it *opens* the object, essentially creating a local copy of the Locator and reassigning its pointers, as shown in Figure 4.3.

At this point, the state of the old Transaction is checked. If it has already committed,

Figure 4.3: Opening a Locator in Object-based TM.



Source: HERLIHY et al. (2003).

the Old Object pointer will be assigned to point to the already committed New Version and the object pointed by New object would be copied, as presented in Figure 4.3. If that Transaction had been aborted, on the other hand, the Old Object would not be reassigned, and the New Object would point to a copy of Old Version instead.

In the case where the old Transaction is still executing, we can conclude that there is a conflict, and one of the transactions will have to be aborted. When a transaction is ready to commit, it simply updates the pointers of the Locators for every object it has modified.

A different line of research, this time on word-based addressing, led into a Transactional Memory design that used a hash table to map from addresses to ownership records (HARRIS; FRASER, 2003), as illustrated by Figure 4.4. In this implementation, ownership is only really acquired during the committing phase, wherein all accessed addresses must become owned by the committing transaction before the data is written to shared memory. If ownership is already acquired for any of the addresses, the transaction must abort.

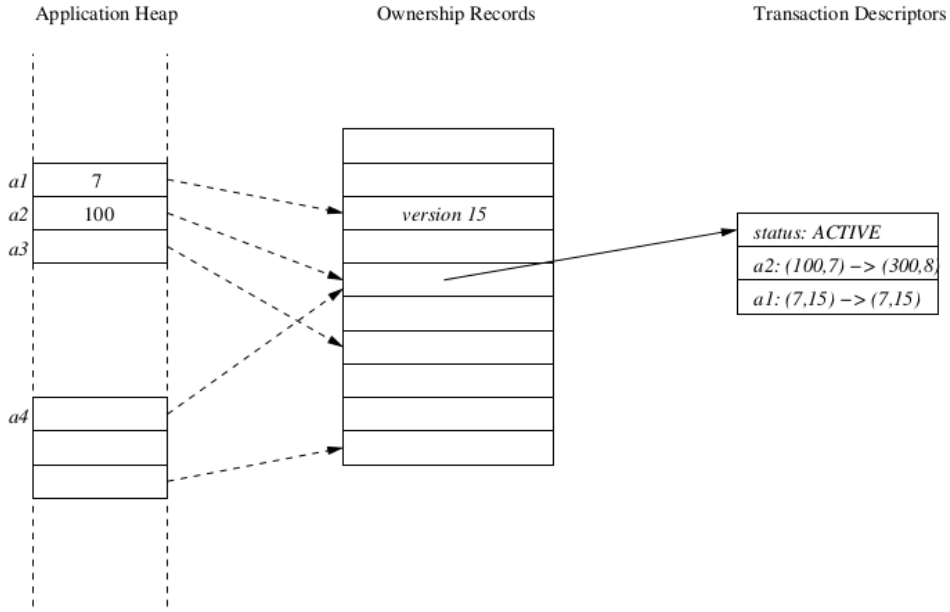
The elimination of the requirement of previous knowledge during TXBEGIN, together with the advent of hash-based Transactional Memory system, have paved the way to newer Software TM designs that implement algorithms that are geared more towards runtime efficiency.

4.1 TL2

Most modern Software TM systems are based on an adaptation of the TL2 algorithm (DICE; SHALEV; SHAVIT, 2006), one of the first designs of a Transactional Memory system that boasted runtime performance that was as efficient as lock-based fine-grained implementations, while being based on fairly straightforward algorithm that only required a simple hardware synchronization facility (such as CAS or LL/SC).

The TL2 TM system provides implementation for both object-based and hash-based ownership semantics, without any difference in its main algorithm. Its design revolves around a global *version-clock*, which is read at the beginning of each transaction, and

Figure 4.4: Hash-table-based TM bookkeeping.



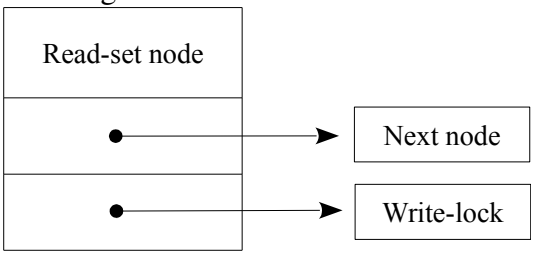
Source: HARRIS; FRASER (2003).

atomically incremented after committing. All memory locations are associated with a *versioned write-lock*. This lock is implemented as an unsigned integer that stores the value of the global version-clock at the moment it was last modified. One of the write-lock's bits is used to indicate whether the associated memory address is currently owned by a transaction.

Since all conflicts must have at least one TXWRITE operation involved, the value of the write-lock for any address can be checked against a local copy of the global version-clock, in order to *validate* the current version of the data. If the current write-lock's version is more recent than the value of the version-clock was at TXBEGIN, we must assume that the address has been modified at some point between the beginning of the transaction and the validation, and abort the current transaction.

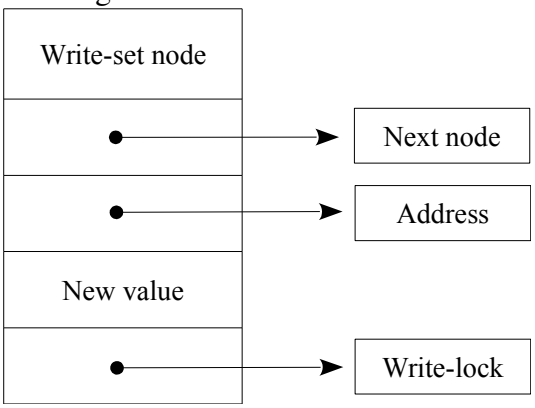
In consideration of allowing the operations required by the TL2 transaction execution algorithm, two data structures must be maintained for each transaction, in the form of read-set and write-set linked list. Figure 4.5 represents a read-set node, which consists of a pointer to the next node and a pointer to the versioned write-lock. The latter pointer is an essential component of validation for previously read data, as it can be used to read the current version and locking status associated with each memory location that accessed by a TXREAD.

Figure 4.5: TL2 read-set node.



A second linked-list is used to represent the execution performed by TXWRITE. Differently from reading operations, writing requires the actual modification of shared data, which TL2 only performs during TXCOMMIT. Because of this design, write-set nodes must also keep track of both the target address and the new value that will be stored in that address at commit-time (as shown in Figure 4.6).

Figure 4.6: TL2 write-set node.



Furthermore, in order to avoid the occurrence of Read-After-Write hazards, TXREAD operations must also check whether the transaction has already written to its target address before reading from shared memory. TL2 uses a bloom filter to determine whether an address has been written to; traversing the write-set list to return the proper value in the affirmative case.

Overall, TL2 proposes an algorithm that displays low overhead for most transactions. The TXBEGIN operation consists in essentially reading the version-clock and locally storing a copy. During the execution, TXWRITE operations append a node to the write-set list. TXREAD operations are validated during execution, catching conflicts as early as possible; if no conflict was detected, the address is optimistically read and a new node is added to the read-set list.

During TXCOMMIT, all locks indicated at the write-set must be acquired, and any failure to do so will abort the transaction. After all locks have been acquired, the global version-clock must be updated, the read set must be validated, and the TXWRITE operations performed during execution must be propagated to shared memory. Finally, each versioned write-lock must be updated with the value of the global version-clock and an unset write-lock bit.

An alternative execution path can be used for read-only transaction. In this scheme, TXBEGIN is still characterized solely by making a copy of the global version-clock. The execution, on the other hand, is performed without the construction of the linked-lists. Rather, each TXREAD will speculatively read from the target address and then immediately validate the operation, by checking the version in its write-lock against the copied version-clock. At any point, failure in validation will abort the transaction. After the execution of a read-only transaction, TXCOMMIT has no further work to perform: reaching this operation while having managed to validate all TXREAD operations guarantees that no conflict has happened, and the transaction can be deemed committed.

4.2 TinySTM

One of the most efficient modern Software TM systems is TinySTM, a successor of TL2 that borrows many concepts from the latter system (FELBER; FETZER; RIEGEL, 2008). Rather than relying solely on the commit-time locking scheme of TL2, TinySTM also implements encounter-time locking, arguing that the early detection of conflicts increases the throughput of transactions, due to less execution of useless work.

Due to the availability of encounter-time locking, TinySTM provides three different methods of TXWRITE execution:

- `WRITE_BACK_CTL`: Acquires write-locks during TXCOMMIT, exactly as in TL2. Version management must be lazy, propagating modifications from the write-set to memory after the locks have been acquired.
- `WRITE_BACK_ETL`: Locks on encounter-time of TXWRITE. Still uses a lazy version management, using the eager locking solely as a mechanism of forcing early conflicts.
- `WRITE_THROUGH`: Locks on encounter-time of TXWRITE, also writing to memory in a scheme of eager version management. In this method, write-sets are substituted by undo-logs, which keep track of the old values that must be restored in the event of a rollback.

In the lazy schemes of version management, the implementation of the write-lock is similar to the one used by TL2, with the exception that instead of always storing a copy of the global version-clock, it stores the pointer to the correspondent write-set node whenever its write-bit is in the *owned* state. This modification allows TinySTM to avoid the complications of Read-After-Write that plague TL2, since TXREAD can easily access the write-set node directly from the owned write-locks.

The eager version management scheme takes advantage of the ability to acquire write-locks at the moment TXWRITE is encountered, using a different algorithm for this operation. Figure 4.7 presents a general view of this eager writing algorithm. In this scheme, a pointer to the owner transaction is stored in the write-lock whenever its write-bit is in the *owned* state.

Figure 4.7: TinySTM eager TXWRITE operation.

```
function txwrite(tx, addr, value)
   lock_addr = &HASH_TABLE[hash(addr)]
   if (*lock_addr & OWNED_BIT)
      owner = *lock_addr & (~OWNED_BIT)
      if (owner == tx)
         *addr = value
      else
         txabort(tx)
   else
      if acquire(tx, lock_addr)
         add_to_buffer(tx, addr, *addr)
         *addr = value
      else
         txwrite(tx, addr, value)
```

Whenever an owned write-lock is encountered, its owner is checked. If the transaction itself is the owner, it can safely modify the address; otherwise, it must either wait for the

write-lock to be released or abort, and by default TinySTM will abort. On the other hand, if the write-lock is not owned, it must be acquired, and an atomic operation (such as CAS) is performed to assign the write-lock with a pointer to the current transaction (with an *owned* write-bit). On success, the current value must be added to an undo-buffer, and the address can then be safely modified.

TinySTM also adopts a hierarchical scheme of TXREAD validation, wherein a second more coarse-grained version of write-locks hash table is initially checked: if no conflict is detected, the fraction of the read-set linked-list that corresponds to those addresses can be skipped; if a conflict is observed, validation proceeds through the traversal of that part of the list, checking against the fine-grained write-locks. This technique can help reduce the overhead of validation for applications in which the frequency of the TXREAD operation far surpasses the frequency of TXWRITE.

The reference implementation of TinySTM also provides a variety of tuning parameters that can be dynamically modified according to heuristics, and can strongly impact the efficiency in the running time of a Transactional Memory application. Some of these parameters are the choice of hash function to map from memory to write-locks, as well as the number of entries in both coarse-grained and fine-grained write-lock hash tables.

4.3 SwissTM

SwissTM is a hash-table-based variant of the TL2 design (DRAGOJEVIć; GUERRAOUI; KAPAŁKA, 2009) which uses a mixed uses a conflict detection scheme known as *mixed invalidation*. In this system, the read/write conflict detection is lazy, while the write/write conflicts are detected during an eager acquisition of write-locks. This behavior aims at avoiding executing transactions that are likely to abort due to write/write conflicts, while reducing the number of aborts that would be caused by eager read/write conflict detection on a transaction that could commit early, in effectively a Write-After-Read context.

In order to support the semantics of mixed invalidation and the entailing memory access algorithms, SwissTM extends the concept of a versioned write-lock hash table, mapping from memory addresses to both a write-lock and a read-lock instead. The semantics of TXWRITE are similar to the ones used in other Software TM systems, as can be seen in Figure 4.8. The main difference from other writing algorithms is regarding the fact that SwissTM acquires locks eagerly during TXWRITE, but needs to keep them in a write buffer to lazily write during TXCOMMIT.

Figure 4.8: SwissTM TXWRITE operation.

```
function txwrite(tx, addr, value)
   owner = WRITE_LOCK[hash(addr)]
   if owner == tx
      update_buffer(tx, addr, value)
   else
      while !acquire_wlock(tx, addr)
         abort_or_wait(tx, addr)
      update_buffer(tx, addr, value)
```

The reading algorithm for SwissTM is significantly different from previous solutions. While it must maintain a read-set linked-list similarly to TL2, the addition of read-locks allows it to busy-wait on these locks during TXREAD until the concurrent transactions

have committed, eliminating the need to abort on some conflicting cases while still preventing the reading of inconsistent data, as the pseudo-code in Figure 4.9 suggests.

Figure 4.9: SwissTM TXREAD operation.

```
function txread(tx, addr)
   owner = WRITE_LOCK[hash(addr)]
   if owner == tx
      return read_from_write_set(tx, addr)
   else
      version1 = READ_LOCK[hash(addr)]
      value = *addr
      version2 = READ_LOCK[hash(addr)]
      if (version1 == LOCKED || version1 != version2)
         return txread(tx, addr)
      add_to_read_set(tx, addr, version1)
      if (!validate(tx,addr) && !try_extend(tx))
         abort(tx)
      else
         return value
```

Similarly to other algorithms, SwissTM will keep a local copy of the global version-lock whenever it starts, and will validate the values it reads against that clock to guarantee the illusion of atomicity regarding the transaction. When that validation fails, however, SwissTM will try to *extend* the transaction: that is, by checking the read-locks of all values in the read-set, if it can guarantee that previously read addresses have not been modified since the time TXBEGIN executed, it can update the local version-clock to the *current* value of the global clock, effectively acting as if the transaction had started its execution after the conflicting memory was modified.

Conflict resolution is performed in SwissTM by a module of the system known as the Contention Manager. Its goal is to merge two different approaches to conflict resolution, deciding between large transactions based on a well-defined order, while avoiding extra overheads for small transactions. This is done by having two different Contention Managers, and having each transaction know its current manager.

Transactions always start out under the Timid contention manager. If they detect a conflict under this state, they will automatically abort and rollback, as fast as possible. On each TXWRITE operation, a local counter is incremented, and the transaction switches to the Greedy conflict manager if a threshold number of accesses has been crossed. Greedy is also used for transactions that have been aborted multiple times.

When resolving the conflict between two Timid transactions, the transaction that detected the conflict will abort. When a conflict between Greedy and Timid transactions is detected, conflict resolution is performed by aborting the Timid transaction. When a conflict is found between two Greedy transactions, a method of establishing a total order among transactions involving a global counter is employed, and this method indicated which transaction should have priority in the conflict.

Consequently, conflict between two Greedy transactions can have multiple possible outcomes. The transaction that detected the conflict can either abort or yield the processor and wait for some time (see Figure 4.8), under the expectation that the conflicting transaction will commit and release the write-lock. Alternatively, the detecting transaction can abort the conflicting transactions and proceed its execution.

Whenever a transaction aborts due to write/write conflicts, it must wait some random time (modulo an upper bound determined in the form of exponential backoff), in an attempt at reducing contention on highly contested memory addresses. In particular, this

behavior avoids repeated aborts caused by the same two transactions contending over the same memory addresses.

4.4 LUTS

In spite of all the recent work on improving the performance of Software Transactional Memory systems, they are still noticeably inefficient under high contention, due to the inherently high overhead of software bookkeeping, which exacerbates the cost of execution and rollback for the conflicting transactions. Most Transactional Memory systems are generally unable to avoid conflicts; they can only decide what transaction to abort during a conflict, and some opt to also enforce a backoff policy to reduce the likelihood of repeating the same conflict.

Rather than waiting for a scenario with a large number of conflicts to apply a palliative measure, a more forward-looking approach would be to predict transactions that are likely to abort and reschedule another transaction that is less likely to conflict in its place. Such anticipatory behavior is the basis for LUTS (Lightweight User-level Transaction Scheduler) (NICÁCIO; BALDASSIN; ARAUJO, 2011), a cooperative user-level thread scheduler that employs heuristics to determine the likelihood of each transaction committing during the next attempted execution.

This design behind LUTS is an attempt at circumventing the problem of false parallelism that results from the spawning of a number of threads that surpasses the number of available processor cores, wherein the excessive number of concurrent transactions is responsible for a higher rate of cache misses, a larger amount of conflicts and the uncontrolled thread preemption driven by a kernel scheduler that has no information regarding the state of the transactional system.

Under LUTS, each program spawns a fixed number of threads that matches the number of cores in the machine. Whenever a system-level thread would have been spawned, LUTS creates an Execution Context Record (ECR), which is a user-level encapsulation of the internal state of the thread. The system-level threads are pinned in their cores, and scheduling of ECRs is performed by the LUTS scheduler, which takes into account the state of the transactions and their conflict estimates when deciding which transaction to schedule.

The scheduler works as an extension of the Transactional Memory system, which must be modified to call the appropriate LUTS subroutines during transaction operations. Inside LUTS, the chosen conflict estimate heuristic is used to decide which user-level thread to schedule. While the scheduler is able to switch execution to a given ECR, its scheduling interface is based on switching to a thread based on its *Lexical Transaction ID* instead (this a number that uniquely identifies the region of the code that corresponds to the transaction; a special ID is used to represent threads that are not executing any transaction), and the two available heuristics will use this interface, relying on the scheduler itself to pick a thread with the given ID for execution.

Internally, the scheduler uses a `ctx_queue` vector of non-blocking concurrent queues, with a queue per Lexical Transaction ID, which is used when extracting a thread for execution based in an ID. LUTS also maintains an `activeTx` vector, indicating what Lexical ID is executing on each core at a given moment, and heuristics may use this information to decide what new ID to schedule based on the likelihood of producing a conflict with the executing threads.

The simplest of the two heuristics available in LUTS is CILUTS, heuristic with neg-

ligible runtime overhead, adapted from an earlier work on transaction conflict prediction (YOO; LEE, 2008b). In this scheme, each Lexical ID is associated with a Conflict Intensity (CI), a number ranging between 0 and 1 that could be interpreted as as estimated probability of conflict with other transactions.

During the startup, all Conflict Intensities are initialized as 0, and their value is updated every time a transaction finishes executing according to the formula:

$$CI' = \begin{cases} \alpha \cdot CI & \text{on commits;} \\ \alpha \cdot CI + (1 - \alpha) & \text{on aborts;} \end{cases}$$

where α determines how much weight past execution will have in the future conflict predictions, decreasing exponentially the older the transactions are. At a transaction's TXBEGIN, CILUTS checks if the CI for its Lexical ID is above a given threshold, and if possible, will schedule in a transaction with lower Conflict Intensity instead.

A more intricate scheduling heuristic is provided by HashLUTS, which maintains a table of conflict estimates based on all concurrently executing transactions, and a `bestTx` table that predicts the best Lexical ID to schedule according to the values in the conflict estimate table. The conflict estimate table is essentially a vector indexed by the Lexical IDs in `activeTx`.

At TXBEGIN, LUTS compares the conflict estimate between the starting transaction and the transactions in other threads; if this estimate is higher in the current thread, a lower estimate thread is scheduled in its place. The conflict estimate table is updated every time the execution of a transaction finishes: if it commits, the conflict estimate is decremented, if it is aborted, the conflict estimate is incremented instead, and the `bestTx` table is updated based on this new estimate. This latter table is central to the design of HashLUTS, which uses it to decide what thread to schedule based on the currently scheduled user-level threads, by taking the Lexical ID from the index of `bestTx` that corresponds to a hash on the active user-level threads, as indicated by the `activeTx` vector.

LUTS supports scheduling under the TinySTM and SwissTM Software TM libraries. Their experiment results showed a significant improvement in the execution time of some of the STAMP and STMBench7 benchmarks, especially in the case of high-contention applications.

5 PROFILING TRANSACTIONAL MEMORY APPLICATIONS

The previous chapters have described the general characteristics of the state-of-the-art implementations and proposals for modern Software and Hardware Transaction Memory systems. In spite of the general goal of making concurrent programming less error-prone than the lock-based solution, and therefore less of a burden on programmers, it is clear that the success of Transaction Memory will also rely on its ability to efficiently execute programs under the paradigm of speculative execution.

In this chapter, we present our proposals for the application of techniques of code profiling for the identification of specific properties in transactional applications, and the subsequent automated tuning of parameters in the TM system based and scheduling of transactions based on conclusions obtained from the collected information.

Initially, we present in Section 5.1 some specific definitions that will be used for the rest of this chapter. In Section 5.2, we explore some related work in the literature that most closely resembles our proposals. Finally, we present in Section 5.3 our proposal and implementation of a tool for automated TM system tuning, and in Section 5.4 our design of a profiling-based scheduling heuristic under LUTS.

5.1 Definitions

When working with Transactional Memory, the word *transaction* is often in the context of different TM systems with slightly different meanings. When working with the profiling of transactional applications, an explicit dissociation of these meanings is oftentimes necessary, as different characterizations of what is a "transaction" will directly affect the interpretation of the collected data.

We say that the Dynamic Instance of a transaction (or just Dynamic Transaction) is the sequence of instructions that is executing between TXSTART and TXCOMMIT (or TXABORT), along with its associated internal data, which is maintained by the Transactional Memory system. When the data accessed during the transaction is already known during compile time, and the whole Signature is not being taken into account, the Static Transaction can be defined similarly. The Signature of a Dynamic Transaction is the list of pointers that comprises its stack backtrace at the beginning of TXSTART (that is, the caller of TXSTART, the caller's caller, and so on, until the `main` function is reached).

Given a set of Dynamic Instances, we define their respective Transaction Type by taking an equivalence relation over this set with regards to their Signature. That is, we say that two Dynamic Instances belong to the same Transaction Type if and only if their Signature is the same. When a Signature of size 1 is assumed, two transactions will be under the same Transaction Type if and only if they would share the same Lexical Transaction ID, if present — that is, two transactions will be considered equivalent when both their

executions originated from the same address of machine code, calling TXBEGIN.

The need for serialization of conflicting transactions gives rise to an important definition that must be taken into account any model of transactional memory behavior: the *serialization rate*, which may be defined as the number of transactions that were serialized over the total number of transactions in a given execution. Serialization rate can also be determined for a particular Transaction Type, as the frequency with which its Dynamic Transactions had to be serialized during the course of an execution. Serialization is such a relevant determiner of the performance of TM applications due to the exclusive-access semantics it imposes.

Even in the cases where serialization is not necessary for the complete execution of conflicting transactions, the overhead of the rollback operation, along with the cost of re-execution, compels for the consideration of the *rollback frequency*, that is, the number of times the instances of a Transaction Type were required to rollback over the total number of times it executed. A high rollback frequency can often be addressed by changing the conflict resolution policy and by a manual restructuring of data access patterns.

5.2 Related Work

There does not seem to exist a great deal of published works on modeling and performance tuning of transactional applications. MARATHE; SCHERER; SCOTT (2004) were among the first to analyze some of the aspects of the design space of Software TM systems. They build on this analysis to develop a new TM system which adapts itself to the workload of the target application, considering the underlying TM system (MARATHE; III; SCOTT, 2005). In spite of the significance of this work, they do not specify a whole performance model, concentrating instead on lock-acquiring semantics for the internal locks used by the TM system.

DALESSANDRO; SCOTT; SPEAR (2010) use transactional memory semantics to model the implementation of traditional synchronization techniques (such as locks, atomic variables and condition variables) in terms of transactions. They define per-thread order relationships that guarantee the property of serializability in the target programs, but performance prediction for TM programs falls outside the scope of their work.

LU; SCOTT (2011) model deterministic semantics for parallel programs based on thread history-based semantics (such as the ones that are used to define memory models). Their modeling consists of a definition of an order relationship for the operations in each thread, interpreting determinism as an equivalence relation over the set of possible parallel program executions. They do not attempt to define performance prediction formulas, though.

The work by PORTER; HOFMANN; WITCHEL (2007) is one of the approaches that most closely resembles our own work. They model the conversion of lock-based programs to transactional ones, helping in the assessment of how suitable is a given lock-based application to be adapted to a Transactional Memory environment. They define for a transaction X the metrics of data dependency (whether $X_R \cap (Y_R \cup X_W) \neq \emptyset$) and conflict density (average of the pairwise number of conflicts between two operations in concurrently executing transactions), which are collected through a profiling execution of a hardware simulator to be used to predict the transactional performance for a given application. This prediction is not applied on any form of automated tuning, however.

While the performance tuning of Transactional Memory programs seems to be an under-researched area, there is a plethora of published tools that are aimed at the debug-

ging and performance tuning of lock-based programs. Some of the most relevant works include YU; RODEHEFFER; CHEN (2005), which implement memory access tracking inside an implementation of the Common Language Infrastructure (MILLER; RAGSDALE, 2003) in order to report suspected race conditions; and work by (OLSZEWSKI; ANSEL; AMARASINGHE, 2009), which uses custom library functions that must be explicitly called by the target application to provide better runtime thread scheduling. Rather than modifying code at runtime, the work by (HARROW, 2000) uses binary instrumentation and a special *pthreads* implementation in order to report possible deadlocks in target code.

There is a growing need to approach applications that rely on Transactional Memory systems the same way one would approach more traditional lock-based applications, using automated tools to help humans develop, optimize and debug parallel code. We are not aware of any published memory model that is able to account for the execution of TM programs and be used as part of an automated application performance tuner. Our work fits that demand by filling this gap in current TM research.

5.3 Improving Syncchar

In spite of all the effort that has been invested into the research of methods to implement efficient Transactional Memory systems, the pursuit of improved performance at a degree that matches the one provided by lock-based applications still may require programmers to manually tinker with the application and TM system parameters in order to optimize their interaction through a trial-and-error approach. Syncchar provides a scheme for predicting the runtime characteristics of TM applications, but still expects programmers to manually configure the TM system based on its predictions.

We propose the extension of Syncchar through the profiling of transaction executions for the prediction of conflict rate, rollback frequency, serialization rate, conflicting Transaction Types, Dynamic Transaction size, number of transactional operations in Dynamic Transaction and hardware capacity overflow in the TM application. These parameters comprise a useful set of data that can be employed by an automated application tuner — which is able to select TM system policies based on these parameters. The following policies can be predicted from the aforementioned parameters:

- The conflict detection policy is a trade-off between either detecting contention eagerly and thus aborting a transaction that would have been able to commit, and detecting it lazily during commit-time and risking having to rollback and re-execute the whole transaction. The *Dynamic Transaction size* should be a good predictor for the better policy: big transactions should abort earlier, to avoid some of the overhead in optimistic execution, while smaller transactions may wish to execute without the overhead of earlier read-set validation and detect their conflicts during commit-time instead. Similarly, a transaction's *rollback frequency*, if high, would imply the need for earlier conflict detection, as measure to minimize the dispute for highly-contented memory addresses. In the case of TM systems providing a hardware variant with an internal software fallback, the frequent occurrence of *hardware capacity overflow* also signalizes an advantage in executing the conflict detection directly in the Software TM subsystem.

- The version management strategy to be employed during eager conflict detection can take one of two forms: eager or lazy locking. Eager locking increases con-

tention due to a higher amount of time holding the locks, but reduces the likelihood of needing to rollback a large amount of operations. A large number of *write* operations increases the likelihood of conflicts, and therefore holds more synergy with an eager locking mechanism. On the other hand, a large number of *read* operations in the set of conflicting Static Transactions implies the need for a lazier method of locking, which would give more window for those transactions to commit. In all cases, a high *conflict rate* should be a warning sign against the increased contention of eager locking.

- When unable to execute a transaction optimistically, serialization must be employed, giving that transaction priority over any conflicting transaction, and preventing other irrevocable transactions from executing. The decision of early serialization can avoid the predictable rollbacks on irrevocable transactions, but will hinder parallelization on cases of spurious serialization. The *serialization rate* of a transaction can safely predict the likelihood of the transaction being irrevocable. Even in the case of more general transactions, a high *rollback frequency* should be met with serialization, as its high contention would probably demand several execution attempts before committing.

- Transaction Memory systems usually allow for the configuration of the conflict detection granularity: a coarser granularity is more likely to yield to the mistake of spurious conflicts, while a finer granularity wastes more memory and is more likely to cause the eviction of a transactional line from cache. In order to predict the appropriate granularity, the *conflict rate* of a transaction could be interpreted as an indicator of the rate for such spurious conflicts; higher conflict rates should be met with a finer granularity, as an attempt to reduce the conflicts. Similarly, a high *serialization rate* should be dealt with by a finer conflict detection granularity, to reduce occurrence of false conflicts under serialization. *Hardware capacity overflow*, on the other hand, is a predictor of scant hardware resources, and should be handled with more coarse transactions.

5.3.1 Implementation

The Syncchar infrastructure is a Simics module that extends the MetaTM transactional architecture by intercepting the transactional hardware instructions (such as TXBEGIN and TXCOMMIT) and checking memory access operations to collect data when executing transactionally. We expand on this implementation by adding calls to the profiler from the MetaTM transactional instruction handler `executeTransactionOp` and the memory access processing code at `ProcessorManager::ProcessMemoryOperation`.

During the installation of the target TM application, it is executed some times under a profiling mode. The profiler collects transactional data from this execution, summarizing it into the aforementioned parameters and permanently storing it into the disk. A script averages the values of the parameters across the executions, building a compact characterization of the application based on the profiling executions.

Since all policies can be distinguished based on a dichotomy on the properties that characterize their behavior, we adopt the application of a system comprised of the linear combination of parameters; whenever the total value resulting from the sum of the weighted parameters surpasses a given threshold, a different policy is applied to the system. The appropriate coefficients and threshold to be used in the equation are derived

from empirical data, identifying the values that yield a greater performance on a set of tuning benchmarks.

5.4 ProfSched: Profile-based scheduling

LUTS was designed to employ a more proactive approach in that it avoids conflicts by scheduling user-level threads whose transactions seem less likely to contend on the same data. During execution, the scheduler builds up on the success or failure of multiple instances of a Lexical Transaction, and slowly starts to characterize them according to their conflict estimates.

This approach, while demonstrably more efficient than the method of straightforward execution utilized by non-scheduling Transactional Memory systems, can be improved through a modification in the scheme of execution: by collecting profiling data during compile-time executions, we can characterize the transactions of a TM application based on a greater variety of data without incurring a prohibitive overhead on the normal execution.

Rather than relying on the characterization of transactions through a single Lexical ID, we propose a scheduling system in which the conflict estimate closely resembles an average on the execution of the instance of Dynamic Transaction itself, accounting for the fact that transactions executed under the same Signature are more likely to be similar than transactions that simply share the same Lexical ID.

We thus propose the use of code profiling as a way of gathering information that can be converted into a reliable conflict estimate among transactions. This information allows us to schedule based on a given transaction at all times during the execution, without having to wait for it to fail multiple times first.

We implement our design through two different build modes for the underlying Software TM library: the *profiling* and the *execution* builds. The profiling build executes the application under LUTS scheduling heuristics, but also dumps information identifying key properties of the executed transactions. At some point after the compilation of a TM application, but before the moment of its execution, we link it with the profiling TM system and execute it.

After a profile dump has been generated, a post-processing script is used to produce a more compact representation of the characteristics of all Dynamic Transactions in the application, and this representation can be seamlessly loaded during execution time to guide our profile-based scheduling heuristic. We implement our heuristic on top of a modified version of LUTS's core scheduling interface, using SwissTM as the Transactional Memory system.

The design of our modifications is divided into two components: TxProf, the profiling module that interacts with both LUTS and the Software TM system to collect data regarding transactional operations and LUTS-based scheduling; and ProfSched, an alternative heuristic to LUTS that relies on profiled data to determine the best scheduling alternative.

5.4.1 Profiling

During the execution of the profiling of an application, we collect data regarding the executing transactions. Every time the application calls a function from the TM interface to perform a transactional operation (such as TXBEGIN, TXREAD or TXWRITE), a function

from the TxProf interface is called to collect this information in memory[1]. When the application has finished executing, the profiling information is dumped into a file.

Internally, TxProf defines a Profiling structure whose fields characterize the properties of the profiled transaction. Table 5.1 presents a high-level view of each field in this structure. During the post-profiling phase, a human-readable file in JSON format is produced from the profile dump, and its entries strongly resemble this internal representation of transactions.

Table 5.1: Fields in a profiling entry.

Field	Description
`this`	A unique ID for each entry.
`precursor`	ID of aborted transaction, if re-executing.
`id_lwp`	Unique thread ID (`schd_getThreadID()`).
`id_ecr`	Unique user-level thread ID (the ECR ID).
`start_bt`	Signature vector (maximum size controlled by `TXPROF_SIZE_STARTBT`).
`oper_n`	Total number of transactional operations.
`oper_fun`	Vector of transactional functions (maximum size controlled by `TXPROF_SIZE_OPERS`).
`oper_enum`	An integer indicating the type of transactional operation in `oper_fun`, such as `READ` or `COMMIT`.
`oper_arg`	The argument to each operation in `oper_fun`.
`tstamps`	Three timestamps, one of them collected at TXBEGIN and the other two before and after committing/aborting.
`enemy`	ID of the conflicting transaction that prompted this transaction to abort, or `0x0`.
`status`	An integer corresponding to one of `BEGAN`, `ABORTED`, `COMMITTED` or `RESTARTED`.

The Signature vector is collected by taking a stack backtrace at the beginning of the TXBEGIN operation (we used GCC's `__builtin_return_address` instead of the more portable `libbacktrace`, because the latter would try to acquire a lock internally). The vector of transactional functions is implemented as the return address for the Software TM library's operation; if this operation is either TXREAD or TXWRITE, the respective argument in `oper_arg` is set to this address. TxProf also collects nanoseconds-precision timestamps, to enable the calculation of the running length for each transaction.

Figure 5.1 presents an algorithm implemented with Transactional Memory (adapted from a similar code in `STAMP/lib/list.c`), along with a simplified representation of the corresponding assembly instructions and their addresses. The algorithm is a list removal operation, consisting in finding the node previous to `elem` in the list and updating its `next` pointer to reference the subsequent node.

Consider an execution of the list removal algorithm where the conditional branch is taken and the transaction executes until TXCOMMIT, but then conflicts with another transaction during commit-time validation. Figure 5.2 represents a possible entry in the JSON

[1] Given the main memory limitations inherent to 32-bit LUTS, we restrict the collected data to an amount of 100 MB, so as to avoid out-of-memory errors in the applications.

Figure 5.1: Example Transactional Memory code.

```
/** Remove element from Singly-Linked List. */
bool list_remove (List* list, void* elem) {
   ListNode *prev, *cur, *next;
   tm_start();                          0x400 49   call  <tm_start>
   prev = find_prev(list, elem);        0x4005c    call  <find_prev>

   if (prev != NULL) {                  0x4006c    cmp   %eax,-0x18(%ebp)
      cur = tm_read(&prev->next);       0x400 79   call  <tm_read>
      next = tm_read(&cur->next);       0x400 89   call  <tm_read>
      tm_write(&prev->next, next);      0x400 a0   call  <tm_write>
      tm_commit();                      0x400 aa   call  <tm_commit>
      return true;                      0x400b5    ret
   }
   tm_commit();                         0x400bc    call  <tm_commit>
   return false;                        0x400c8    ret
}
```

profiling file, created during such execution (for simplicity, all transactional operations inside `find_prev` have been omitted).

Figure 5.2: Example TxProf JSON entry.

```
"entry": {
   "this": 0xd66e0,
   "precursor": 0xd42c8,
   "id_lwp": 2,
   "id_ecr": 17,
   "start_bt": [0x400 49, 0x58814, 0x54dd1, 0x541b5],
   "oper_n": 4,
   "oper_name": ["read",     "read",     "write", "commit"],
   "oper_fun": [0x400 79,    0x400 89,   0x400 a0,  0x400 aa],
   "oper_arg": [&prev->next, &cur->next, &prev->next,  0x0],
   "enemy": 0xd68e8,
   "status": 203
}
```

The value corresponding to `this` is the unique identifier for this entry, while the presence of a non-zero `precursor` indicates that this is a re-execution of a previously failed transaction, under the specified identifier. The `id_lwp` and `id_ecr` fields specify the system-level and user-level (ECR) threads that were tasked with the execution of this transaction, respectively.

Note how the top of the `start_bt` stack contains the address of the instruction that called the function `tm_start`, while the next address in the stack corresponds to the caller of `list_remove`, and then its parent caller. For each operation named in `oper_name`, the respective address in `oper_fun` corresponds to the callers of transactional operations inside the conditional branch, with the function argument indicated in `oper_arg` (here represented as their corresponding high-level C code for didactic purposes). Since the TXCOMMIT operation does not take any target argument, its corresponding argument is `0x0`.

Finally, the presence of an `enemy` identifier indicates that this transaction has terminated due to a conflict with the transaction represented by the transaction matching the profile identifier. This is corroborated by the `status` field, which contains the value that corresponds to the `RESTARTED` state, indicating the outcome of the execution of this transaction.

It is important to observe that, while the backtrace from TXBEGIN in `start_bt` by itself would already contain enough information to characterize instances of Dynamic

Transactions according to their Signature, TxProf also collects additional information that allow it to predict how long a transaction will take to execute and how likely it is to be involved in a conflict, permitting more advanced scheduling heuristics than the ones employed by LUTS.

In order to collect the needed profiling data, TxProf subroutines must be called at startup, as well as during each transactional operation performed by the Software TM system. During the system-level thread initialization, a call to `txprof_thread_init` ensures the allocation of memory and initialization of a thread-local linked-list of empty Profiling entries, which will be filled during the execution of the program — thread-local instances are used because they greatly reduce the costs of creating a new Profiling entry during TXBEGIN, whereas a global linked-list would require some method of synchronization during the acquisition of each new entry.

Table 5.2 gives a brief description of some of the main subroutines that comprise the TxProf module, indicating where they are called and the profiling data that they collect. These calls must be explicitly added to both the Software TM system and LUTS at the appropriate places.

Table 5.2: Summary of TxProf subroutines.

TxProf Subroutine	Description
`txprof_thread_init()`	Called by `luts_init` once for each thread during the application startup.
`txprof_start(*p)`	Called inside TXBEGIN before the execution of each transaction. Creates a new Profiling entry, setting its `start_bt`.
`txprof_read_beg(p,addr)`	Called inside TXREAD before reading transactionally from an address. Collects both the caller's address and the value of `addr`.
`txprof_write_beg(p,addr)`	Called inside TXWRITE before writing transactionally to an address. Collects both the caller's address and the value of `addr`.
`txprof_commit_beg(p)`	Called inside TXCOMMIT before trying to commit a transaction. Collects a timestamp.
`txprof_commit_end(p)`	Called inside TXCOMMIT after a transaction has successfully committed. Collects a timestamp and sets the `COMMITTED` status.
`txprof_abort_beg(p,enemy)`	Called inside TXABORT before starting to rollback the aborted transaction. Collects a timestamp and sets the `enemy` transaction pointer.
`txprof_abort_end(p)`	Called inside TXABORT, right before long-jumping to TXBEGIN for re-execution. Collects a timestamp and sets the `ABORTED` status.
`txprof_file_dump(file)`	Called after the application has executed. Dumps the collected profile information.

5.4.2 Post-Profiling

The data that is collected during the profiling phase is then processed by scripts so that useful information can be extracted in a compact representation. This concise character-

ization of the TM application can be directly manipulated by the post-profiling scripts to provide additional runtime information to the transaction scheduler, without any manual intervention to adapt the original application. Three files are output by the post-profiling processor, and then loaded by the scheduler during the execution, as described below:

Running time for each transaction.

During the post-profiling processing, the length of execution (obtained from the timestamps) is gathered for all instances of each Transaction Type. This information is then summarized into a hash table, mapping from a Transaction Type to the average running length of its Dynamic Transactions.

A plain-text representation of this hash table is stored in the `avgruntime.txt` file, to be loaded during execution time into the `commit_length` hash table — which maps from the Signature of a Transaction Type to its average running length (in nanoseconds), serving to determine whether a Dynamic Transaction should be treated by the scheduler as a short transaction or a long one.

Number of transactional operations.

Similarly to the collected hash table on the average running time for each Transaction Type, the average number of transactional operations can be also be gathered into a `avgtxops.txt` file, built from the value of the `oper_n_real` Profiling field.

During execution, the file is loaded into a `txopers_length` hash table inside the scheduler, which can be used as as estimate for the number of transactional operations that will be executed by a Dynamic Transaction — a predictor of probability of conflict with other transactions, as well as a predictor for transaction operation overhead and likelihood of required serialization.

Graph of conflicts.

A directed graph of conflicts is built by the post-profiler based on the properties summarized from Dynamic Instances. Each node in this graph represents a Transaction Type, and these nodes store information regarding Transaction Types based on the execution of their respective instances during profiling:

- `signature`, the Signature for this Transaction Type.
- `n_runs`, the total number of times the transaction instances have executed.
- `n_aborts`, the number of executions of that have failed to commit.

Post-profiling also identifies the interaction between Dynamic Transactions through the `enemy` indication in Profiling instances, and this is reflected in the edges of this graph. An edge from T_a to T_b stores information regarding the past interaction between instances of those Transaction Types:

- `n_conflicts`, which is the number of times an instance of T_a has conflicted with an instance of T_b.
- `wasted_time`, which is the total amount of time (in nanoseconds) that has been lost executing instances of T_a which then conflicted with T_b.

- `n_shared_runs`, which is the number of times T_a executed, except for the cases where it conflicted with some transaction that was not T_b. That is, given the set R of nodes adjacent to T_a,

$$n\_shared\_runs(T_a, T_b) = n\_runs(T_a) - \sum_{\substack{T_x \in R \\ x \neq B}} n\_conflicts(T_a, T_x).$$

  Here, we assume that the probability of T_a conflicting with T_b is the same as it would be if T_a did not conflict with any other transaction — which is why we explicitly ignore those data points. This is a somewhat strong assumption that may lead to an overestimation of the actual probability of conflict in some cases, which can make the scheduler perform needlessly conservative re-scheduling decisions, but at the same time it permits the prediction of interaction between of each pair of transactions without full knowledge of how they all affect each other pairwise.

At the beginning of the execution, the described graph is loaded into memory. The set of nodes is implemented as an array indexed by a hash on the Signature of each transaction. Edges (T_a, T_b) are represented through adjacency lists at node T_a, implemented as a hash table on the Signature of T_b, which maps into a C language structure with a field for each information in the edge.

During execution, the scheduler can update this information based on transactions that are currently executing, effectively combining data from profiling with locally acquired execution-time data. However, since this incurs in extra overhead, extra care must be taken to only update information for sufficiently large transactions, lest this bookkeeping become a performance bottleneck.

Once all information has been loaded at the beginning of the execution, an estimate of the Conflict Probability between two transactions can be easily and efficiently derived from this graph: the probability of T_a conflicting with T_b is

$$CP(T_a, T_b) = n\_conflicts(T_a, T_b)/n\_shared\_runs(T_a, T_b).$$

This value is a number between 0 and 1 which estimates the probability of a conflict based on the frequency with which an instance of Transaction Type T_a has conflicted with a concurrent instance of T_b during the profiling phase.

5.4.3 Scheduling

During the execution of the TM application, the data structures loaded from the post-profiled files can be accessed by a scheduling module, to inform its decisions regarding who to schedule at each processor core. Rather than internally characterizing a transaction through its Lexical Transaction ID, as previous works have done (DRAGOJEVIć; GUERRAOUI; KAPAłKA, 2009; NICÁCIO; BALDASSIN; ARAUJO, 2011), we use the full Signature for a transaction to determine when it should be scheduled.

We adapt the core scheduling capabilities of our system from prior work by LUTS (described in Section 4.4), and extend the scheduler by applying a profiling-based scheduling heuristics, using a more refined set of data structures. Due to the more extensive scheduling requirements of ProfSched, the following data structures are required:

- A hash-based mapping keeps track of the currently executing ECR thread for each system-level thread. This is conceptually similar to the `activeTx` list under LUTS,

but ProfSched maintains a pointer to the actual ECR context that is executing, rather than just storing the Lexical ID for its transaction. Given the ECR context object, we can access the Signature, as well as any other data collected during the execution of the current Dynamic Transaction, if any.

- A Signature-indexed hash table of queues, collecting the ECR threads that are waiting for execution for each Transaction Type, as well as the threads that are not currently executing any transactions. An ECR thread can be popped from one of these queues when the scheduler decides to schedule a thread executing a given Transaction Type.

Although LUTS must rely on the running time of a transaction just aborted to predict whether its re-execution should be classified as either a small or a long transaction, ProfSched is able to take advantage of the values from the `commit_length` hash table, which allows it to predict the execution length of any transaction at the moment it starts executing for the first time[2].

Long Transactions

When a transaction is deemed a Long Transaction, we compare its Conflict Estimate with its Expected Gain to decide whether it can be executed right away. Given the list A of currently active transactions, the Conflict Estimate for a transaction T is an approximation of the expected amount of time that will have been wasted if it conflicts with transaction A_i and needs to be restarted:

$$CE(T) = \sum_{0 \le i < |A|} \left\{ \prod_{0 \le j < i} \Big[1 - CP(T, A_j) \Big] \cdot CP(T, A_i) \cdot wasted\_time(T, A_i) \right\}.$$

For a precise calculation of the Conflict Estimate, the elements of A should be sorted in ascending order of $wasted\_time(T, A_i)$, as otherwise the product of previous probabilities may yield a spurious value.

We define the Expected Gain for a transaction T as the probability of successfully executing it (that is, the probability of not conflicting with transactions of any of the concurrent threads), scaled by its expected $commit\_length$:

$$EG(T) = \prod_{0 \le i < |A|} \big[1 - CP(T, A_i) \big] \cdot commit\_length(T).$$

This value estimates the amount of progress that will be made in overall execution if T commits after being scheduled right away.

Figure 5.3 presents the algorithm we developed to calculate the Conflict Estimate and the Expected Gain for a given ECR thread. Note that, while the definition of Conflict Estimate would at first suggest an implementation of $\Theta(k^2)$ running time for k available processor cores, we use a partial calculation of the inner product of Conflict Probabilities to derive an efficient $\Theta(k)$ algorithm, effectively proportional to the constant number of processor cores.

Whenever a non-transactional thread starts executing a transaction T that is predicted to be a Long Transaction, ProfSched calculates both its Conflict Estimate and its Expected

[2]We classify a transaction as *long* if its expected commit length surpasses 10^4 ns. LUTS uses a similar classification when automatically alternating between heuristics at the $5 \cdot 10^4$ ns threshold.

Figure 5.3: Calculating the Conflict Estimate and the Expected Gain.

```
function calculate_CE_EG(t)
   sum = 0; partial_prod = 1
   for t2 in active_ecr_threads
      if executing_transaction(t2)
         p = conflict_probability(t, t2)
         sum = sum + partial_prod * p * wasted_nsecs(t, t2)
         partial_prod = partial_prod * (1 – p)
   ConflictEstimate = sum
   ExpectedGain = partial_prod * commit_length(t)
```

Gain, and compares them to decide whether it would be more worthwhile to attempt its execution. ProfSched uses a static value of $\rho = 0.5$ to weigh the difference in impact between these two values, and if

$$CE(T) > \rho \cdot EG(T),$$

another transaction is scheduled in the current system-level thread, and the ECR thread in which T was executing is pushed into the queue of waiting threads to be later re-scheduled under more favorable circumstances.

Small Transactions

The above scheme does not work well for small transactions, since the overhead of calculating conflict probabilities outweighs any gains obtained from achieving less conflicts. As in LUTS, we use the CILUTS heuristic to determine whether a short transaction is expected to commit, and only schedule a transaction if it has a sufficiently small Conflict Intensity — otherwise, another transaction is chosen in its place.

6 EXPERIMENTAL RESULTS

Chapter 5 presented two different proposals involving the application of profiling techniques towards the optimization of the execution of Transactional Memory applications. This chapter presents an evaluation of our implementations for both the Syncchar-based Hardware Transactional Memory profiling approach (Section 6.1) and the LUTS-based user-thread transaction scheduler (Section 6.2).

6.1 Syncchar Results

Execution was performed on the Simics 3.0.17 simulator for the x86 architecture (with simulator modules compiled under GCC 4.6.4), using eight $3\,\mathrm{GHz}$ processors, a 4-way $16\,\mathrm{KBd}$ $16\,\mathrm{KBi}$ L1 cache under XMESI (1-cycle hit), 8-way $4\,\mathrm{MB}$ L2 cache under the regular MESI (16-cycle hit) and $1\,\mathrm{GB}$ main memory (200-cycle hit). We used a modified 32-bit Linux kernel version 2.6.16, which is the one originally used during the development of Syncchar[1], and STAMP benchmark version 0.9.9 using simulator workloads (all compiled under GCC 4.5.3).

When working with a simulator, the physical time it takes to perform the simulation can often be orders of magnitude higher than the simulated time. While this physical time property is not the focus of our profiling concerns, it is still relevant in matters of practicality of use. When disabling custom XMESI cache-coherency protocol from MetaTM, a single-processor execution of the lock-based STAMP SSCA2 benchmark is simulated in about an hour of physical time, taking 6.12×10^8 cycles to execute under simulated time. Enabling the full transactional support, the execution time of transactional SSCA2 can range from about 8 hours, for the simulation of single-processor architectures, to around 4 days of physical time at the 8-processor Syncchar configuration.

In spite of months of attempted execution, Syncchar has proven to be an unstable infrastructure, and our initial implementation of the profiling tool has never been properly executed in a way that reliably produced profiling data. We list some of the major problems encountered during the execution of Syncchar in Appendix A.

Given the underlying problems encountered in simulating Transactional Memory hardware, we deduce an overall inability to obtain profiling information from this framework. It relies on a simulator which, despite appearing to seamlessly handle transactional semantics on a single-processor execution, fails to execute the appropriate semantics on a multicore simulation, thus hiding parallel concurrency from any profiling tools.

[1] We also tried other versions of the Linux kernel, in particular the version 3.5, which was the newest version at the time we proposed this work. Simics 3.0.17 seemed to crash during the boot of kernel 3.5, and even older kernels were too incompatible with the kconfig parameterization for Syncchar's kernel and failed to execute `update-initramfs`.

6.2 ProfSched Results

We compare the results from executions under ProfSched with the results from both the unmodified SwissTM Software TM system as the Baseline, as well as the LUTS scheduler using its default heuristic. Execution was performed for all three systems on two different target platforms, described below.

The first platform is `viking`, a 2-processor machine with 4 cores per processor (multithreading was disabled), each processor executing at a frequency of $2.4\,\mathrm{GHz}$ on a 64-bit x86 architecture. Its memory hierarchy includes a $32\,\mathrm{KB}$d $32\,\mathrm{KB}$i L1 cache, $256\,\mathrm{KB}$ L2 cache, $8\,\mathrm{MB}$ L3 cache and $24\,\mathrm{GB}$ main memory. The operating system is 64-bit Ubuntu 12.04, with Linux kernel 3.2.

The second platform is `turing`, a 4-processor machine with 8 cores per processor (multithreading was disabled), with each processor executing at a frequency of $2\,\mathrm{GHz}$ on a 64-bit x86 architecture. Its memory hierarchy includes a $32\,\mathrm{KB}$d $32\,\mathrm{KB}$i L1 cache, $256\,\mathrm{KB}$ L2 cache, $18\,\mathrm{MB}$ L3 cache and $128\,\mathrm{GB}$ main memory. The operating system is 64-bit Ubuntu 12.04, with Linux kernel 3.2.

All of SwissTM, LUTS, TxProf, ProfSched and the STAMP benchmarks were compiled with GCC 4.6.3 (`-O1` optimization level), similar to the configurations adopted when LUTS was tested at (NICÁCIO; BALDASSIN; ARAUJO, 2011). Due to inherent limitations of LUTS's internal stack management algorithm, everything was compiled under 32-bit x86 mode.

Table 6.1 presents the running time achieved by ProfSched on STAMP benchmarks for 1, 2, 4, and 8 threads. Each entry in the table was generated through $n = 30$ executions of the given benchmark, and its content reflects the average running time (in seconds) for those executions and the sample standard deviation (using Bessel's $n - 1$ correction).

As can be seen in the table, not all executions finish successfully. While we have managed to understand and correct some of the problems that plagued earlier benchmark executions, in spite of all the time that has been spent on debugging the internals of LUTS along with the original authors, one application still fails to finish execution after weeks, while another one finishes its execution producing wrong results, and a further one goes astray at a pointer's dereference and attempts to access invalid memory.

While the executions of all benchmarks finish correctly under the Baseline, failures on the part of ProfSched always entail a similar failure when executing under LUTS. Recently, the authors of LUTS have conceded that the results presented in (NICÁCIO; BALDASSIN; ARAUJO, 2011) were produced after sieving out cases of stack overflow and wrong output, and that the original STAMP benchmarks also presented some intermittent invalid memory accesses that were never fully understood.

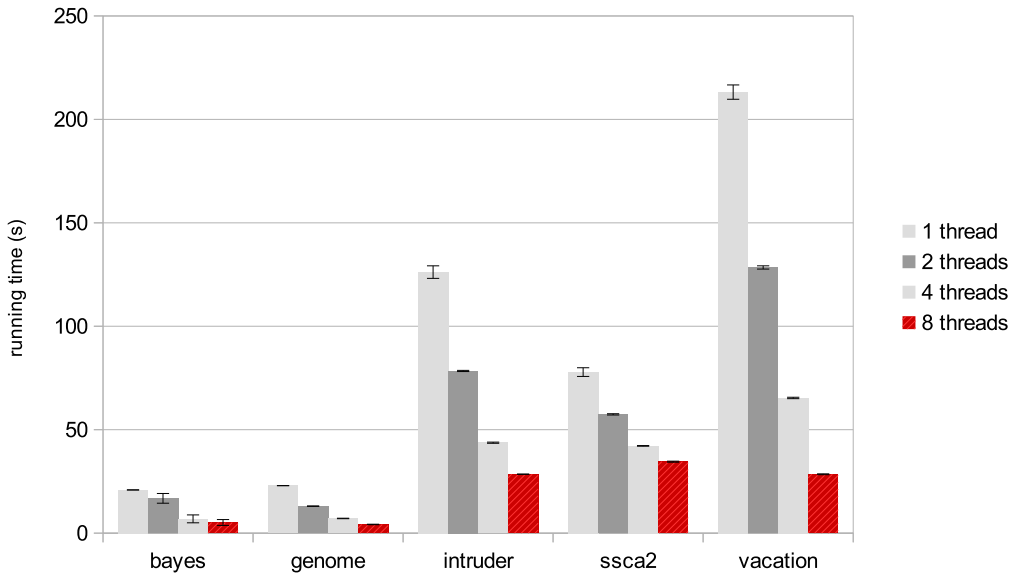
Table 6.1: Running times (in seconds) $\pm$ standard deviation on `viking`.

Benchmark	1 thread	2 threads	4 threads	8 threads
bayes	20.87 ± 0.02	16.80 ± 1.20	6.89 ± 0.97	5.14 ± 0.72
genome	22.95 ± 0.01	13.04 ± 0.06	7.08 ± 0.06	4.19 ± 0.06
intruder	126.18 ± 1.55	78.36 ± 0.14	43.70 ± 0.14	28.48 ± 0.04
kmeans	56.23 ± 1.37	∞	∞	∞
labyrinth	114.52 ± 2.20	`Wrong`	`Wrong`	`Wrong`
ssca2	77.82 ± 1.07	57.40 ± 0.15	42.16 ± 0.08	34.53 ± 0.13
vacation	213.17 ± 1.77	128.44 ± 0.40	65.33 ± 0.18	28.45 ± 0.10
yada	`SegFault`	`SegFault`	`SegFault`	`SegFault`

While the prospect presented above might seem rather bleak, it is important to notice that the applications that executed completely *did* calculate their answer correctly. Furthermore, the most insidious errors we have found were all related to the way the compiler generated optimized code — such as the omission of stack pointers (which corrupted LUTS's ECR thread checkpointing) and the miscompilation of code where the compiler assumed that a volatile-qualified variable was not accessed and did not allocate space for it on the stack. When compiling the code without optimizations, all errors disappear, further advancing the hypothesis that the failed executions happened due to no particular fault in ProfSched, rather being the outcome of ill-defined behavior in either LUTS or even the STAMP benchmarks themselves.

Figure 6.1 presents the average running time for the successful executions on `viking`, along with their corresponding 95% confidence interval (calculated from the aforementioned sample standard deviation, assuming that the random disturbances in running time follow a Gaussian distribution). As can be readily seen, the most significant effects on running time occur at the longest benchmarks, `intruder` and `vacation`, while more modest improvements in running time are observed in other benchmarks. Similar results can be seen in Figure 6.2 for executions on `turing`, though the scalability of STAMP for this machine's 32 cores is somewhat less pronounced, particularly in the case of smaller benchmarks. Variation in the running times is more visible in this machine, and affects the small benchmark `bayes` to a greater extent.

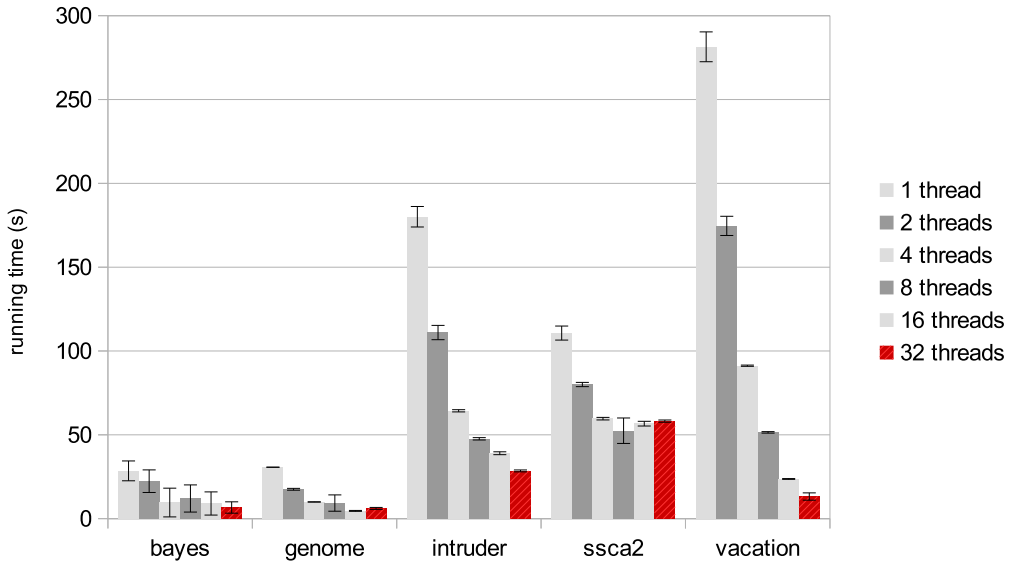
Figure 6.1: Running times for ProfSched on `viking`.



Since one of the key motivations to using Transactional Memory is the ability to scale an application to a high number of processor cores in spite of memory access conflicts (a factor that would serialize execution under the lock-based paradigm), different systems of speculative execution must be compared under this parameter. Figure 6.3 presents the base-2 logarithm of the speedup for `viking` $\left({}^{T_1}/_{T_8}\right)$ and `turing` $\left({}^{T_1}/_{T_{32}}\right)$ running the *light* STAMP workloads (see Appendix B for a table of raw speedup values).

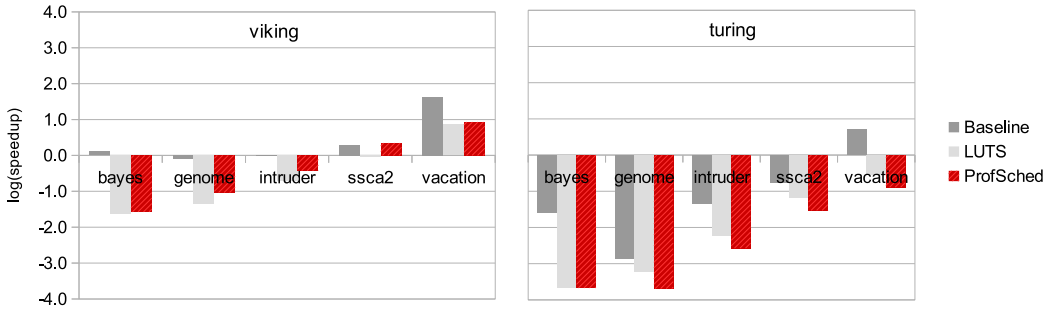
As a ratio of two numbers, the speedup from two different executions can be readily compared through a visual inspection whenever presented logarithmically. For example,

Figure 6.2: Running times for ProfSched on `turing`.



in the *light* execution at `viking`, the speedup of ProfSched for `vacation` is inversely proportional to its speedup for `genome`. Overall, the executions with *light* workload had worsening speedups, particularly for LUTS and ProfSched, through even the Baseline scaled poorly under these workloads.

Figure 6.3: Log-speedup for the *light* workload.



The executions with *medium* workloads, on the other hand, show more mixed speedup results, as can be seen in Figure 6.4. In this case, most executions on `viking` yielded an improvement in performance for its 8 cores, while the equivalent execution in `turing`'s 32 cores maintained a slowdown when compared to the sequential execution — albeit a less pronounced performance impact than the one observed with the *light* workloads.

As can be seen, scheduling techniques have not really been effective for the *light* and *medium* workloads, and this is actually not very surprising: even the Baseline results do not scale at all during most of these executions, since the overhead introduced by the software TM system is more than enough to outweigh any performance gain obtained through parallelism. When even the Baseline fails to keep up in execution time, any scheduling heuristic is doomed to fail as well.

Figure 6.4: Log-speedup for the *medium* workload.

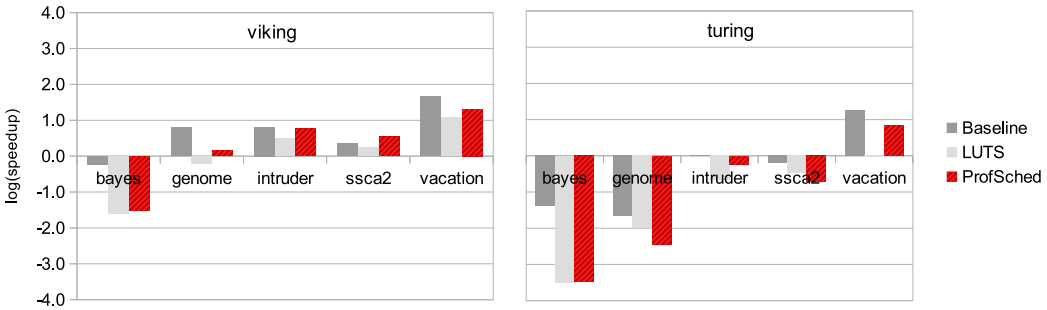
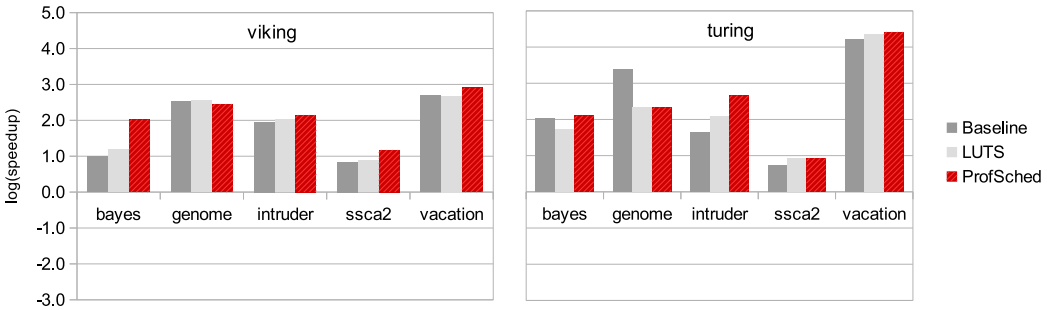


Figure 6.5 presents the logarithm of speedups for *heavy* STAMP workloads. As can be observed in this figure, Baseline (SwissTM) presents above-constant speedup for all executions, allowing LUTS and ProfSched to similarly succeed in their scheduling. Note that, in spite of the dampening visual effect of log-scales, in most cases ProfSched presents a visually higher speedup than both the Baseline and LUTS.

Figure 6.5: Log-speedup for the *heavy* workload.



The consistent improvement of ProfSched over the alternatives can be more thoroughly recognized at Tables 6.2 and 6.3. In these tables, the raw (non-logarithmic) speedup for all *heavy* executions is presented for both `viking` and `turing`. Note how, in both architectures, the geometric mean of ProfSched speedups is higher than the one achieved by Baseline and LUTS. In fact, while the geometric mean for LUTS on `turing` is lower than Baseline, ProfSched is reasonably higher.

Table 6.2: Speedup under *heavy* workload on `viking`.

Speedup	Baseline	LUTS	ProfSched
bayes	1.98	2.28	4.06
genome	5.77	5.86	5.47
intruder	3.88	4.07	4.43
ssca2	1.79	1.85	2.25
vacation	6.43	6.40	7.49
Geometric mean	3.48	3.65	**4.41**

Table 6.3: Speedup under *heavy* workload on `turing`.

Speedup	Baseline	LUTS	ProfSched
bayes	4.08	3.34	4.30
genome	10.51	5.07	5.03
intruder	3.10	4.25	6.33
ssca2	1.65	1.90	1.90
vacation	18.37	20.50	21.34
Geometric mean	5.26	4.89	**5.61**

Given this data, we can readily see the improvement of ProfSched over the alternative methods. On `viking`, it is 26.71% more efficient than Baseline and 20.83% more efficient than LUTS; on `turing`, the results are more modest, though still positive: 6.67% more efficient than Baseline and 14.73% more efficient than LUTS.

Not shown in the data is the overhead of TxProf collection, which accounts for about 10–20% of extra running time. Since this overhead is only incurred once, during the profiling phase, and it is equally distributed among all executing transactions, we deemed it a reasonable cost — unlikely to skew the results of profiling or to excessively burden the process of installing the application into the target system.

While the ineffectiveness of state-of-the-art debuggers (such as `gdb` and `valgrind`) in dealing with speculative execution have hindered the uncovering of the problems in the failing STAMP applications, the proposed approach for profile-based scheduling in Transactional Memory systems looks sound, and achieves significantly better results than previously observed.

As the data implies, scheduling of transactional applications still greatly depends the efficiency of the underlying software TM system in dealing with the application workloads: whenever the overhead of speculation obstructs the performance of parallel execution, the efficiency of scheduling techniques is thoroughly hindered. On the other hand, whenever the software TM system presents a more favorable speedup scaling with the number of concurrent threads, our scheduling heuristic is able to consistently turn that behavior to its advantage, further improving the running time of the transactional applications.

7 FINAL CONSIDERATIONS

The ultimate success of Transactional Memory will depend on whether it can actually supersede traditional lock-based techniques in terms of performance and ease of use. Current solutions still often demand that programmers manually tinker with the transactional application and the speculative system to optimize the execution.

We build on a preexisting work devoted to Hardware Transactional Memory simulation and performance modeling, as well as a user-level thread scheduler for Software Transactional Memory systems, in both cases adding a phase of profile gathering to the installation of an application in a given environment, and tuning the application based on conclusions automatically derived from this collected data.

Our first proposal is an extension to the Syncchar Hardware TM infrastructure that relies on the nature of processor simulation to seamlessly acquire detailed access pattern data during a profiling execution, which can then be used to predict the most favorable policies for the underlying Transactional Memory system when actually executing the application. This prediction can be used to automatically tune the speculative system to the particular application and runtime environment.

We also propose a method of low-overhead profiling and summarization of data access patterns from Dynamic Transactions in Software TM system applications, and a profile-guided scheduling of user-level threads based on the characteristics inferred for the currently executing transactions. We write a profile-based scheduler based on a state-of-the-art scheduling interface, extending it to use our novel representation of transactions through their Signature, so that the conflict estimates closely resemble an average on the execution of Dynamic Instances, accounting for the fact that transactions executed under the same Signature are more likely to be similar than transactions that simply share a superficial lexical identification.

7.1 Contributions

On the whole, despite all the work that can be found in the literature regarding automatic optimization of lock-based applications, the field of Transactional Memory is still comparatively recent, and generally lacks the basic tools aimed at debugging and performance tuning. Our work on automated profile-based tuning of applications fills a part of this gap by designing, implementing and evaluating methods of automatically optimizing the execution of transactional applications.

In the case of our first proposal, we implement a profiling tool on top of Syncchar and perform experiments on it, reaching the unfortunate conclusion that the framework is not reliably usable as a basis for our profiling techniques, as it is built upon a simulator which fails to execute the appropriate transactional semantics on a multicore simulation, which

in turn hinders parallel concurrency from our profiling tool.

Regarding the second proposal, we implement and compare it against the state-of-the-art Transactional Memory systems SwissTM and LUTS on two different architectures, achieving a improvement in execution time of around 27% over SwissTM and 15% over LUTS. This improvement over the execution time of Transactional Memory applications will greatly favor the future adoption of this method of synchronization in high-performance contexts.

7.2 Future Work

The ProfSched scheduling approach treats the Conflict Probability $CP(T_a, T_b)$ uniformly even when the Transaction Types are the same (that is, $T_a = T_b$). In the cases where this probability is too high, a scheduler could attempt to differentiate between Dynamic Instances, blocking the execution of one of the threads and re-scheduling another thread less likely to conflict in its place, perhaps even applying a more specific method of conflict resolution for these "self-conflicting" cases.

ProfSched schedules transactions based on a comparison between their Conflict Estimate and their Expected Gain, weighted by a static value. While this approach is a reasonable early method of comparison, it is possible that a more refined comparison might yield a more unbiased scheduling when there are threads with low Expected Gain that have a high Conflict Estimate, whose execution is all but prevented by this heuristic. A variable weighting based on past execution, or even some randomization, may lead to a scheduling that is more fair towards all transactions and avoids occasional incidents of performance degradation due to excessive serialization.

While the profiling information collected allows for the scheduling of transactions based on a more complete view of the whole application, it is important to notice that our work on LUTS focuses only on this scheduling, and does not take full advantage of all optimizations that could be performed based on this profiling. For example, the choice of conflict detection mechanism for an application would likely benefit from the knowledge of what Transaction Types are expected to be executing at a given moment. A post-profiling summarization of what Signatures dominate the execution at each point should allow the dynamic tuning of the underlying Software TM system based on these parameters.

APPENDIX A: A BRIEF GUIDE TO SYNCCHAR

A.1 Introduction

The Syncchar TM infrastructure (described in Sections 3.3 and 5.3) can be downloaded from `http://z.cs.utexas.edu/users/osa/metatm/releases/1.2/osa.img.bz2`, which contains the image file with the in-simulator file system. After downloading Syncchar and setting up a Simics workspace, the Linux kernel and Syncchar modules must be compiled and copied to the image file.

The C++ compiler must be GCC, version 4.5. Versions older than 4.1 are known for miscompiling the `map_demap` interface (an internal C-language representation of memory mapping in Simics), and newer versions are too strict when searching for `#include` directives, failing to compile the Syncchar C++ module. On some versions of GCC, the `arch/i386/kernel/Makefile` must be modified to use `-m32` instead of `-m x86_i386`, and the `static` qualifier must be removed from mutex functions in `kernel/mutex.c`. The Linux kernel can then be easily compiled with `linux32 make HOSTCC=gcc-4.5`, and a similar operation can be used to compile each of the STAMP benchmarks. The Syncchar utility script `scripts/cp_linux` can then be used to copy the kernel and the benchmarks into the image file.

A.2 Inconveniences

When executing `make clean; make` under Syncchar, care must be taken, as some modules in the `sws/modules` directory must be manually compiled with independent `make` calls before the execution of Syncchar. Failure to do so will be silently ignored and cause segmentation faults in the target simulated programs (not during the loading of Syncchar modules).

Simics uses an image file to represent the file system. Since this file must be mounted by the root user, Syncchar scripts must often be executed with super-user privileges, and will produce files that will be further used by Syncchar. At any error during the Simics boot-up, one should check whether any simulator files are owned by the root, and change to the appropriate user instead. In particular, the `sws/grub.simics` file must not have been removed by the Syncchar scripts.

During the course of our work, we encountered two MetaTM assertions that fail under the name `OSA_ASSERT FAILURE`. One of them checked whether the `pTx` transaction pointer has been relieved after TXCOMMIT, which must be fixed by the addition of a call to the MetaTM magic instruction `OSA_MAGIC(OSA_KERNEL_BOOT)` in the Linux kernel's `start_kernel` at `init/main.c`. The other assertion was caught because the current thread disappeared from MetaTM's local version of process table; the authors of Sync-

char seemed to think it wasn't really a problem, since Syncchar didn't rely on it, and the solution consisted in commenting out this check.

One of the most insidious error messages consisted in the repeated printing of the line `System call 119 made inside a user tx`, seemingly stuck in a loop. This may be a result of a signal handler being called synchronously from inside a transaction. The authors of Syncchar were able to reproduce this, and introduced a few extra hooks into MetaTM from the kernel to properly handle context switching during a transaction. Their SVN code has since been updated with a fixed version of MetaTM.

The aforementioned error message could also be caused by scheduling of threads with negative PIDs (we always saw the PID `-12347`, deterministically), caused by the kernel not properly informing MetaTM of process creation and context switches, corrupting its memory instead. This problem is handled with a tricky set of special Simics instructions that must be called at key points in the kernel. At the time we experienced this error, the authors of MetaTM were able to identify the missing instruction and updated the SVN code appropriately.

Some common warning messages, such as `odd magic breakpoint 0x2058a0` and `xcpt#: 7 No translation addr: 0x39f90417` can be safely ignored. Other more serious error messages, such as the sporadic failure of execution of TM benchmarks with signal 11 (presumably Segmentation Fault) still require further looking into. The Syncchar authors have not been able to identify and reproduce the source of this error, but its terse `PROCESS EXIT FAILURE: (SIGNAL: 11, CODE: 0)` error message implies that any measurement of execution time would just be meaningless.

APPENDIX B: RAW DATA

With the purpose of furthering future comparisons, we present the full execution data collected from ProfSched (omitted from Chapter 5.4 in the interest of brevity).

B.1 Running time

Table B.1 is similar to Table 6.1, but contains data from `turing`. This data is presented more visually at Figure 6.2.

Table B.1: Running times (in seconds) $\pm$ standard deviation at `turing`.

Benchmark	1 thread	2 threads	4 threads
bayes	28.50 ± 3.02	22.32 ± 3.44	9.60 ± 4.37
genome	30.66 ± 0.02	17.51 ± 0.27	9.96 ± 0.04
intruder	180.12 ± 3.12	111.01 ± 2.18	64.32 ± 0.34
ssca2	110.68 ± 2.13	80.00 ± 0.64	59.63 ± 0.38
vacation (high)	281.45 ± 4.54	174.66 ± 2.92	91.21 ± 0.18
Benchmark	8 threads	16 threads	32 threads
bayes	12.03 ± 4.13	9.04 ± 3.52	6.63 ± 1.73
genome	9.29 ± 2.48	4.61 ± 0.14	6.10 ± 0.29
intruder	47.61 ± 0.36	39.02 ± 0.44	28.46 ± 0.28
ssca2	52.43 ± 3.88	56.65 ± 0.72	58.18 ± 0.34
vacation (high)	51.44 ± 0.23	23.67 ± 0.12	13.19 ± 1.11

Not shown in the Tables 6.1 and B.1 is the overhead of TxProf collection, which accounts for about 10–20% of extra running time. Since this overhead is only incurred once, during the profiling phase, and it is equally distributed among all executing transactions, we deem it a reasonable cost — unlikely to skew the results of profiling or to excessively burden the process of installing the application into the target system.

B.2 Speedup

Tables B.2 and B.3 are similar to Tables 6.2 and 6.3, but using *light* STAMP workloads. This data is presented more visually at Figure 6.3. In the same vein, Tables B.4 and B.5 represent the *medium* STAMP workloads. This data is presented more visually at Figure 6.4.

Table B.2: Speedup under *light* workload at `viking`.

Speedup	Baseline	LUTS	ProfSched
bayes	1.08	0.33	0.34
genome	0.93	0.39	0.49
intruder	1.01	0.61	0.74
ssca2	1.21	0.98	1.27
vacation	3.08	1.82	1.91
Geometric mean	1.31	0.67	**0.78**

Table B.3: Speedup under *light* workload at `turing`.

Speedup	Baseline	LUTS	ProfSched
bayes	0.33	0.08	0.08
genome	0.14	0.11	0.08
intruder	0.39	0.21	0.17
ssca2	0.59	0.44	0.35
vacation	1.64	0.69	0.54
Geometric mean	0.45	0.22	**0.18**

Table B.4: Speedup under *medium* workload at `viking`.

Speedup	Baseline	LUTS	ProfSched
bayes	0.85	0.33	0.35
genome	1.74	0.87	1.12
intruder	1.76	1.41	1.71
ssca2	1.28	1.18	1.45
vacation	3.16	2.10	2.49
Geometric mean	1.60	1.00	**1.19**

Table B.5: Speedup under *medium* workload at `turing`.

Speedup	Baseline	LUTS	ProfSched
bayes	0.38	0.09	0.09
genome	0.31	0.25	0.18
intruder	1.01	0.63	0.84
ssca2	0.87	0.72	0.61
vacation	2.38	0.98	1.80
Geometric mean	0.76	0.40	**0.43**

REFERENCES

ALDEA, S.; LLANOS, D.; GONZÁLEZ-ESCRIBANO, A. Towards a Compiler Framework for Thread-Level Speculation. In: PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2011 19TH EUROMICRO INTERNATIONAL CONFERENCE ON, 2011. **Proceedings...** [S.l.: s.n.], 2011. p.267 –271.

ANSARI, M.; KOTSELIDIS, C.; WATSON, I.; KIRKHAM, C.; LUJáN, M.; JARVIS, K. Lee-TM: a non-trivial benchmark suite for transactional memory. In: IN PROCEEDINGS OF THE 8TH INTERNATIONAL CONFERENCE ON ALGORITHMS AND ARCHITECTURES FOR PARALLEL PROCESSING, ICA3PP, 2008. **Proceedings...** [S.l.: s.n.], 2008.

ASANOVIC, K.; BODIK, R.; DEMMEL, J.; KEAVENY, T.; KEUTZER, K.; KUBIATOWICZ, J.; MORGAN, N.; PATTERSON, D.; SEN, K.; WAWRZYNEK, J.; WESSEL, D.; YELICK, K. A view of the parallel computing landscape. **Commun. ACM**, New York, NY, USA, v.52, p.56–67, October 2009.

BAEK, W.; MINH, C. C.; TRAUTMANN, M.; KOZYRAKIS, C.; OLUKOTUN, K. The OpenTM Transactional Application Programming Interface. In: PROCEEDINGS OF THE 16TH INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURE AND COMPILATION TECHNIQUES, 2007, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2007. p.376–387. (PACT '07).

BAILEY, D. H.; BARSZCZ, E.; BARTON, J. T.; BROWNING, D. S.; CARTER, R. L.; DAGUM, L.; FATOOHI, R. A.; FREDERICKSON, P. O.; LASINSKI, T. A.; SCHREIBER, R. S.; SIMON, H. D.; VENKATAKRISHNAN, V.; WEERATUNGA, S. K. The NAS parallel benchmarks: summary and preliminary results. In: PROCEEDINGS OF THE 1991 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 1991, New York, NY, USA. **Proceedings...** ACM, 1991. p.158–165. (Supercomputing '91).

BAUGH, L.; NEELAKANTAM, N.; ZILLES, C. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In: PROCEEDINGS OF THE 35TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 2008, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p.115–126. (ISCA '08).

BHOWMIK, A.; FRANKLIN, M. A general compiler framework for speculative multithreading. In: PROCEEDINGS OF THE FOURTEENTH ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, 2002, New York, NY, USA. **Proceedings...** ACM, 2002. p.99–108. (SPAA '02).

BIENIA, C.; KUMAR, S.; SINGH, J. P.; LI, K. The PARSEC benchmark suite: characterization and architectural implications. In: PROCEEDINGS OF THE 17TH INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.72–81. (PACT '08).

BOBBA, J.; MOORE, K. E.; VOLOS, H.; YEN, L.; HILL, M. D.; SWIFT, M. M.; WOOD, D. A. Performance pathologies in Hardware Transactional Memory. In: IN: PROCEEDINGS OF THE 34RD ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE. INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 2007. **Proceedings...** [S.l.: s.n.], 2007. p.81–91.

CASCAVAL, C.; BLUNDELL, C.; MICHAEL, M.; CAIN, H. W.; WU, P.; CHIRAS, S.; CHATTERJEE, S. Software transactional memory: why is it only a research toy? **Commun. ACM**, New York, NY, USA, v.51, p.40–46, November 2008.

CHAPMAN, B.; JOST, G.; PAS, R. v. d. **Using OpenMP**: portable shared memory parallel programming (scientific and engineering computation). [S.l.]: The MIT Press, 2007.

CHAUDHRY, S.; CYPHER, R.; EKMAN, M.; KARLSSON, M.; LANDIN, A.; YIP, S.; ZEFFER, H.; TREMBLAY, M. Rock: a high-performance SPARC CMT processor. **Micro, IEEE**, [S.l.], v.29, n.2, p.6 –16, march-april 2009.

CHOQUETTE, J. H.; TENE, G.; NORMOYLE, K. **Speculative multiaddress atomicity**. US Patent 7,376,800. Azul Systems, Inc.

CHRISTIE, D.; CHUNG, J.-W.; DIESTELHORST, S.; HOHMUTH, M.; POHLACK, M.; FETZER, C.; NOWACK, M.; RIEGEL, T.; FELBER, P.; MARLIER, P.; RIVIèRE, E. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In: PROCEEDINGS OF THE 5TH EUROPEAN CONFERENCE ON COMPUTER SYSTEMS, 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.27–40. (EuroSys '10).

CHUNG, J.; YEN, L.; DIESTELHORST, S.; POHLACK, M.; HOHMUTH, M.; CHRISTIE, D.; GROSSMAN, D. ASF: AMD64 extension for lock-free data structures and transactional memory. In: MICROARCHITECTURE (MICRO), 2010 43RD ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON, 2010. **Proceedings...** [S.l.: s.n.], 2010. p.39 –50.

CLICK, C. **Azul's Experiences with Hardware Transactional Memory**. Accessed on 2012-06-13., http://sss.cs.purdue.edu/projects/tm/tmw2010/talks/Click-2010_TMW.pdf.

CYTRON, R.; FERRANTE, J.; ROSEN, B. K.; WEGMAN, M. N.; ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, New York, NY, USA, v.13, n.4, p.451–490, 1991.

DALESSANDRO, L.; CAROUGE, F.; WHITE, S.; LEV, Y.; MOIR, M.; SCOTT, M. L.; SPEAR, M. F. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In: PROCEEDINGS OF THE SIXTEENTH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING

LANGUAGES AND OPERATING SYSTEMS, 2011, New York, NY, USA. **Proceedings...** ACM, 2011. p.39–52. (Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)).

DALESSANDRO, L.; SCOTT, M. L.; SPEAR, M. F. Transactions as the foundation of a memory consistency model. In: PROCEEDINGS OF THE 24TH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING, 2010, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2010. p.20–34. (DISC'10).

DAVE, C.; BAE, H.; MIN, S.-J.; LEE, S.; EIGENMANN, R.; MIDKIFF, S. Cetus: a source-to-source compiler infrastructure for multicores. **Computer**, [S.l.], v.42, n.12, p.36 –42, dec. 2009.

DICE, D.; LEV, Y.; MOIR, M.; NUSSBAUM, D. Early experience with a commercial hardware transactional memory implementation. In: PROCEEDINGS OF THE 14TH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.157–168. (ASPLOS '09).

DICE, D.; SHALEV, O.; SHAVIT, N. Transactional locking II. In: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING, 2006, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2006. p.194–208. (DISC'06).

DOHERTY, S.; GROVES, L.; LUCHANGCO, V.; MOIR, M. Towards formally specifying and verifying transactional memory. **Formal Aspects of Computing**, [S.l.], v.25, n.5, p.769–799, 2013.

DRAGOJEVIć, A.; FELBER, P.; GRAMOLI, V.; GUERRAOUI, R. Why STM can be more than a research toy. **Commun. ACM**, New York, NY, USA, v.54, n.4, p.70–77, Apr. 2011.

DRAGOJEVIć, A.; GUERRAOUI, R.; KAPAŁKA, M. Stretching transactional memory. In: PROCEEDINGS OF THE 2009 ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.155–165. (PLDI '09).

FELBER, P.; FETZER, C.; RIEGEL, T. Dynamic Performance Tuning of Word-based Software Transactional Memory. In: PROCEEDINGS OF THE 13TH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.237–246. (PPoPP '08).

GAONA, E.; ABELLáN, J. L.; ACACIO, M. E.; FERNáNDEZ, J. Deploying Hardware Locks to Improve Performance and Energy Efficiency of Hardware Transactional Memory. In: PROCEEDINGS OF THE 26TH INTERNATIONAL CONFERENCE ON ARCHITECTURE OF COMPUTING SYSTEMS, 2013, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2013. p.220–231. (ARCS'13).

GARZARáN, M. J.; PRVULOVIC, M.; LLABERÍA, J. M.; VIñALS, V.; RAUCHWERGER, L.; TORRELLAS, J. Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors. In: PROCEEDINGS OF THE 9TH INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 2003,

Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2003. p.191–. (HPCA '03).

GARZARáN, M. J.; PRVULOVIC, M.; LLABERíA, J. M.; VIñALS, V.; RAUCHWERGER, L.; TORRELLAS, J. Software Logging under Speculative Parallelization. In: IN WORKSHOP ON MEMORY PERFORMANCE ISSUES, IN CONJUNCTION WITH ISCA-28, 2001. **Proceedings...** [S.l.: s.n.], 2001.

GOODMAN, J. R. Using cache memory to reduce processor-memory traffic. In: PROCEEDINGS OF THE 10TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 1983, New York, NY, USA. **Proceedings...** ACM, 1983. p.124–131. (ISCA '83).

GOPAL, S.; VIJAYKUMAR, T.; SMITH, J.; SOHI, G. Speculative Versioning Cache. In: PROCEEDINGS OF THE 4TH INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 1998, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1998. p.195–.

GUERRAOUI, R.; KAPALKA, M.; VITEK, J. STMBench7: a benchmark for software transactional memory. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.41, p.315–324, March 2007.

GUO, R.; AN, H.; DOU, R.; CONG, M.; WANG, Y.; LI, Q. LogSPoTM: a scalable thread level speculation model based on transactional memory. In: COMPUTER SYSTEMS ARCHITECTURE CONFERENCE, 2008. ACSAC 2008. 13TH ASIA-PACIFIC, 2008. **Proceedings...** [S.l.: s.n.], 2008. p.1 –8.

HARRIS, T.; FRASER, K. Language support for lightweight transactions. In: ACM SIGPLAN NOTICES, 2003. **Proceedings...** [S.l.: s.n.], 2003. v.38, n.11, p.388–402.

HARRIS, T.; LARUS, J. R.; RAJWAR, R. **Transactional Memory, 2nd edition**. [S.l.]: Morgan & Claypool Publishers, 2010. (Synthesis Lectures on Computer Architecture).

HARRIS, T.; MARLOW, S.; PEYTON-JONES, S.; HERLIHY, M. Composable memory transactions. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2005, New York, NY, USA. **Proceedings...** ACM, 2005. p.48–60. (PPoPP '05).

HARROW, J. J. Runtime Checking of Multithreaded Applications with Visual Threads. In: PROCEEDINGS OF THE 7TH INTERNATIONAL SPIN WORKSHOP ON SPIN MODEL CHECKING AND SOFTWARE VERIFICATION, 2000, London, UK, UK. **Proceedings...** Springer-Verlag, 2000. p.331–342.

HEBER, T.; HENDLER, D.; SUISSA, A. On the impact of serializing contention management on STM performance. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v.72, n.6, p.739–750, June 2012.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture**: a quantitative approach. 4th.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

HERLIHY, M.; LUCHANGCO, V.; MOIR, M.; SCHERER III, W. N. Software Transactional Memory for Dynamic-sized Data Structures. In: PROCEEDINGS OF THE

TWENTY-SECOND ANNUAL SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 2003, New York, NY, USA. **Proceedings...** ACM, 2003. p.92–101. (PODC '03).

HERLIHY, M.; MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. In: PROCEEDINGS OF THE 20TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 1993, New York, NY, USA. **Proceedings...** ACM, 1993. p.289–300. (ISCA '93).

HOFMANN, O. S.; ROSSBACH, C. J.; WITCHEL, E. Maximum Benefit from a Minimal HTM. In: PROCEEDINGS OF THE 14TH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.145–156. (ASPLOS XIV).

Intel Corporation. **Transactional Synchronization in Haswell**. Accessed on 2013-10-24, http://software.intel.com/en-us/blogs/2012/02/07 /transactional-synchronization-in-haswell.

Intel Corporation. **Intel 64 and IA-32 architectures optimization manual**. Accessed on 2014-01-07, http://www.intel.com/content/dam/www/public/us/en /documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

KNIGHT, T. An architecture for mostly functional languages. In: PROCEEDINGS OF THE 1986 ACM CONFERENCE ON LISP AND FUNCTIONAL PROGRAMMING, 1986, New York, NY, USA. **Proceedings...** ACM, 1986. p.105–112. (LFP '86).

KUMAR, S.; CHU, M.; HUGHES, C. J.; KUNDU, P.; NGUYEN, A. D. Hybrid transactional memory. In: PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING (PPOPP'06), 2006. **Proceedings...** [S.l.: s.n.], 2006. p.209–220.

LARUS, J. R.; RAJWAR, R. **Transactional Memory**. [S.l.]: Morgan & Claypool, 2006.

LIU, W.; TUCK, J.; CEZE, L.; AHN, W.; STRAUSS, K.; RENAU, J.; TORRELLAS, J. POSH: a tls compiler that exploits program structure. In: PROCEEDINGS OF THE ELEVENTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2006, New York, NY, USA. **Proceedings...** ACM, 2006. p.158–167. (PPoPP '06).

LU, L.; SCOTT, M. L. Toward a formal semantic framework for deterministic parallel programming. In: PROCEEDINGS OF THE 25TH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING, 2011, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2011. p.460–474. (DISC'11).

MAGNUSSON, P. S.; CHRISTENSSON, M.; ESKILSON, J.; FORSGREN, D.; HÅLLBERG, G.; HöGBERG, J.; LARSSON, F.; MOESTEDT, A.; WERNER, B. Simics: a full system simulation platform. **Computer**, Los Alamitos, CA, USA, v.35, p.50–58, February 2002.

MANNARSWAMY, S. S.; GOVINDARAJAN, R. Reconciling transactional conflicts with compiler's help. In: CGO, 2012. **Proceedings...** ACM, 2012. p.53–62.

MARATHE, V. J.; III, W. N. S.; SCOTT, M. L. Adaptive Software Transactional Memory. In: IN PROC. OF THE 19TH INTL. SYMP. ON DISTRIBUTED COMPUTING, 2005. **Proceedings...** [S.l.: s.n.], 2005. p.354–368.

MARATHE, V. J.; SCHERER, W. N.; SCOTT, M. L. Design tradeoffs in modern software transactional memory systems. In: PROCEEDINGS OF THE 7TH WORKSHOP ON WORKSHOP ON LANGUAGES, COMPILERS, AND RUN-TIME SUPPORT FOR SCALABLE SYSTEMS, 2004, New York, NY, USA. **Proceedings...** ACM, 2004. p.1–7. (LCR '04).

MARTIN, M. M. K.; SORIN, D. J.; BECKMANN, B. M.; MARTY, M. R.; XU, M.; ALAMELDEEN, A. R.; MOORE, K. E.; HILL, M. D.; WOOD, D. A. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.33, p.92–99, November 2005.

MILLER, J. S.; RAGSDALE, S. **The Common Language Infrastructure Annotated Standard**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

MINH, C. C.; CHUNG, J.; KOZYRAKIS, C.; OLUKOTUN, K. STAMP: stanford transactional applications for multi-processing. In: WORKLOAD CHARACTERIZATION, 2008. IISWC 2008. IEEE INTERNATIONAL SYMPOSIUM ON, 2008. **Proceedings...** [S.l.: s.n.], 2008. p.35 –46.

MOIR, M.; MOORE, K.; NUSSBAUM, D. The Adaptive Transactional Memory Test Platform: a tool for experimenting with transactional code for rock. In: TRANSACT '08: 3RD WORKSHOP ON TRANSACTIONAL COMPUTING, 2008. **Proceedings...** [S.l.: s.n.], 2008.

MOORE, K.; BOBBA, J.; MORAVAN, M.; HILL, M.; WOOD, D. LogTM: log-based transactional memory. In: HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 2006. THE TWELFTH INTERNATIONAL SYMPOSIUM ON, 2006. **Proceedings...** [S.l.: s.n.], 2006. p.254 – 265.

NICÁCIO, D.; BALDASSIN, A.; ARAUJO, G. LUTS: a lightweight user-level transaction scheduler. In: ICA3PP (1), 2011. **Proceedings...** Springer, 2011. p.144–157. (Lecture Notes in Computer Science, v.7016).

OANCEA, C. E.; MYCROFT, A.; HARRIS, T. A lightweight in-place implementation for software thread-level speculation. In: PROCEEDINGS OF THE TWENTY-FIRST ANNUAL SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES, 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.223–232. (SPAA '09).

OLSZEWSKI, M.; ANSEL, J.; AMARASINGHE, S. Kendo: efficient deterministic multithreading in software. In: PROCEEDINGS OF THE 14TH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.97–108. (ASPLOS '09).

PERI, S.; VIDYASANKAR, K. Correctness of concurrent executions of closed nested transactions in transactional memory systems. **Theoretical Computer Science**, [S.l.], v.496, p.125–153, 2013.

PORTER, D. E.; HOFMANN, O. S.; ROSSBACH, C. J.; BENN, A.; WITCHEL, E. Operating System Transactions. In: PROCEEDINGS OF THE ACM SIGOPS 22ND SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.161–176. (SOSP '09).

PORTER, D. E.; HOFMANN, O. S.; WITCHEL, E. Is the optimism in optimistic concurrency warranted? In: PROCEEDINGS OF THE 11TH USENIX WORKSHOP ON HOT TOPICS IN OPERATING SYSTEMS, 2007, Berkeley, CA, USA. **Proceedings...** USENIX Association, 2007. p.1:1–1:6. (HOTOS'07).

PORTER, D. E.; WITCHEL, E. Understanding transactional memory performance. In: IEEE INTERNATIONAL SYMPOSIUM ON PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE, 2010. **Proceedings...** IEEE Computer Society, 2010. p.97–108.

RAMADAN, H. E.; ROSSBACH, C. J.; PORTER, D. E.; HOFMANN, O. S.; BHANDARI, A.; WITCHEL, E. MetaTM/TxLinux: transactional memory for an operating system. **IEEE Micro**, Los Alamitos, CA, USA, v.28, n.1, p.42–51, 2008.

RAUBER, T.; RÜNGER, G. **Parallel Programming**: for multicore and cluster systems. [S.l.]: Springer, 2010. I-X, 1-455p.

RENAU, J.; FRAGUELA, B.; TUCK, J.; LIU, W.; PRVULOVIC, M.; CEZE, L.; SARANGI, S.; SACK, P.; STRAUSS, K.; MONTESINOS, P. **SESC simulator**. http://sesc.sourceforge.net.

ROSSBACH, C. J. **Hardware transactional memory**: a systems perspective. [S.l.]: The University of Texas at Austin, 2011.

SHAVIT, N.; TOUITOU, D. Software transactional memory. In: PROCEEDINGS OF THE FOURTEENTH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 1995, New York, NY, USA. **Proceedings...** ACM, 1995. p.204–213. (PODC '95).

SPEAR, M. F.; DALESSANDRO, L.; MARATHE, V. J.; SCOTT, M. L. A comprehensive strategy for contention management in software transactional memory. In: PROCEEDINGS OF THE 14TH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.141–150. (PPoPP '09).

STEFFAN, J. G.; COLOHAN, C. B.; ZHAI, A.; MOWRY, T. C. A scalable approach to thread-level speculation. In: PROCEEDINGS OF THE 27TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 2000, New York, NY, USA. **Proceedings...** ACM, 2000. p.1–12. (ISCA '00).

STEFFAN, J. G.; COLOHAN, C.; ZHAI, A.; MOWRY, T. C. The STAMPede approach to thread-level speculation. **ACM Trans. Comput. Syst.**, New York, NY, USA, v.23, p.253–300, August 2005.

VIKTOR LEIS ALFONS KEMPER, T. N. Exploiting Hardware Transactional Memory in Main-Memory Databases. In: IEEE INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 30., 2014, Chicago, IL, USA. **Proceedings...** [S.l.: s.n.], 2014. (ICDE 2014).

WALIULLAH, M.; STENSTROM, P. Removal of Conflicts in Hardware Transactional Memory Systems. **International Journal of Parallel Programming**, [S.l.], v.42, n.1, p.198–218, 2014.

WANG, A.; GAUDET, M.; WU, P.; AMARAL, J. N.; OHMACHT, M.; BARTON, C.; SILVERA, R.; MICHAEL, M. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In: PROCEEDINGS OF THE 21ST INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 2012, New York, NY, USA. **Proceedings...** ACM, 2012. p.127–136. (PACT '12).

WANG, C.; CHEN, W.-Y.; WU, Y.; SAHA, B.; ADL-TABATABAI, A.-R. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In: PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION, 2007, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2007. p.34–48. (CGO '07).

WARG, F.; STENSTROM, P. Dual-thread speculation: a simple approach to uncover thread-level parallelism on a simultaneous multithreaded processor. **Int. J. Parallel Program.**, Norwell, MA, USA, v.36, p.166–183, April 2008.

WOO, S. C.; OHARA, M.; TORRIE, E.; SINGH, J. P.; GUPTA, A. The SPLASH-2 programs: characterization and methodological considerations. In: PROCEEDINGS OF THE 22ND ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 1995, New York, NY, USA. **Proceedings...** ACM, 1995. p.24–36. (ISCA '95).

WU, C.; LIAN, R.; ZHANG, J.; JU, R.; CHAN, S.; LIU, L.; FENG, X.; ZHANG, Z. An Overview of the Open Research Compiler. In: EIGENMANN, R.; LI, Z.; MIDKIFF, S. (Ed.). **Languages and Compilers for High Performance Computing**. [S.l.]: Springer Berlin, Heidelberg, 2005. p.922–922. (Lecture Notes in Computer Science, v.3602).

YOO, R. M.; LEE, H.-H. S. Helper Transactions: enabling thread-level speculation via a transactional memory system. In: WORKSHOP ON PARALLEL EXECUTION OF SEQUENTIAL PROGRAMS ON MULTI-CORE ARCHITECTURES (PESPMA), HELD IN CONJUNCTION WITH THE 35TH ACM/IEEE INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE (ISCA), 2008. **Proceedings...** [S.l.: s.n.], 2008. (PESPMA '08).

YOO, R. M.; LEE, H.-H. S. Adaptive Transaction Scheduling for Transactional Memory Systems. In: PROCEEDINGS OF THE TWENTIETH ANNUAL SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES, 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.169–178. (SPAA '08).

YOURST, M. T. PTLsim: a cycle accurate full system x86-64 microarchitectural simulator. In: PERFORMANCE ANALYSIS OF SYSTEMS SOFTWARE, 2007. ISPASS 2007. IEEE INTERNATIONAL SYMPOSIUM ON, 2007. **Proceedings...** [S.l.: s.n.], 2007. p.23 –34.

YU, Y.; RODEHEFFER, T.; CHEN, W. RaceTrack: efficient detection of data race conditions via adaptive tracking. In: PROCEEDINGS OF THE TWENTIETH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 2005, New York, NY, USA. **Proceedings...** ACM, 2005. p.221–234. (SOSP '05).

ZHAI, A.; STEFFAN, J. G.; COLOHAN, C. B.; MOWRY, T. C. Compiler and hardware support for reducing the synchronization of speculative threads. **ACM Trans. Archit. Code Optim.**, New York, NY, USA, v.5, p.3:1–3:33, May 2008.