Architectural Support for High-Performance Hardware Transactional Memory Systems

by

Marc Lupon i Navazo

Submitted in Fulfillment of the
Requirements for the Degree

Doctor of Philosophy
Programa de Doctorat: Arquitectura de Computadors

Supervised by

Grigorios Magklis
Antonio González Colás

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

November 8, 2011

”Per als meus pares, Isa i Emili, que sempre m’han ajudat, inclús sense saber-ho”

Acknowledgments

Són moltes les persones les quals m'han ajudat durant aquesta tesi; estic convençut que sense elles no hauria acabat aquest llarg camí. No seria just si no comencés pel meu director, en Grigoris, no només per guiar-me durant tots aquests anys sinó perquè s'ha preocupat per mi en tot moment, i encara avui és capaç d'aguantar les meves "neures". També vull agrair al meu co-director, l'Antonio, per donar-me l'oportunitat de fer el doctorat, aconsellar-me en tot moment i ensenyar-me què és la recerca.

M'agradaria seguir per tota la gent que m'ha fet més entretinguda la vida al campus, ja sigui prenent cafès al bar, posant la decoració de Nadal a la sala o jugant a futbol als migdies. Òscar, Iñaki, Beacco, Enric, Niko, Javi, Demos, Gemma, Xavi, Renné, Manu, Miquel, Marc, Eduard, membres de la sala C6-E208, jugadors del DEE+, gent del gimnàs i resta de companys d'ARCO, d'IBRC i del DAC: moltes gràcies a tots! També vull agrair al Mark D. Hill i a tota la gent del Wisconsin Multifacet Project per fet la meva estància a Madison més agradable.

Per acabar, vull donar les gràcies a les meves dues famílies, la de sang i l'adoptiva. Tinc la sort d'haver crescut amb en Ferran, Dani, Vidi, Gus, Maria, Cesc, Puyi, Torra, Rafa, Maurici, Núria, Ricard, Oriol, Albert i molts altres que segur m'oblido. Amb amics com ells es pot superar qualsevol entrebanc, i tota anècdota es torna un record inesborrable.

Finalment vull donar les gràcies als meus avis; ells em van ensenyar la importància d'intentar ser sempre treballador i bona persona. També a la meva germana Anna, ella és la única que em coneix millor que jo mateix. I als meus pares, Isa i Emili, perquè sense ells no seria ni una desena part de la persona que sóc avui en dia.

Abstract

Parallel programming presents an efficient solution to exploit future multicore processors. Unfortunately, traditional programming models depend on programmer's skills for synchronizing concurrent threads, which makes the development of parallel software a hard and error-prone task. In addition to this, current synchronization techniques serialize the execution of those critical sections that conflict in shared memory and thus limit the scalability of multi-threaded applications.

Transactional Memory (TM) has emerged as a promising programming model that solves the trade-off between high performance and ease of use. In TM, the system is in charge of scheduling transactions (atomic blocks of instructions) and guaranteeing that they are executed in isolation, which simplifies writing parallel code and, at the same time, enables high concurrency when atomic regions access different data. Among all forms of TM environments, Hardware TM (HTM) systems is the only one that offers fast execution at the cost of adding dedicated logic in the processor.

Existing HTM systems suffer considerable delays when they execute complex transactional workloads, especially when they deal with large and contending transactions because they lack adaptability. Furthermore, most HTM implementations are *ad hoc* and require cumbersome hardware structures to be effective, which complicates the feasibility of the design. This thesis makes several contributions in the design and analysis of low-cost HTM systems that yield good performance for any kind of TM program.

Our first contribution, FASTM, introduces a novel mechanism to elegantly manage speculative (and already validated) versions of transactional data by slightly modifying on-chip memory engine. This approach permits fast recovery when a transaction that fits in private caches is discarded. At the same time, it keeps non-speculative values in software, which allows in-place

memory updates. Thus, FASTM is not hurt from capacity issues nor slows down when it has to undo transactional modifications.

Our second contribution includes two different HTM systems that integrate deferred resolution of conflicts in a conventional multicore processor, which reduces the complexity of the system with respect to previous proposals. The first one, FUSETM, combines different-mode transactions under a unified infrastructure to gracefully handle resource overflow. As a result, FUSETM brings fast transactional computation without requiring additional hardware nor extra communication at the end of speculative execution. The second one, SPECTM, introduces a two-level data versioning mechanism to resolve conflicts in a speculative fashion even in the case of overflow.

Our third and last contribution presents a couple of truly flexible HTM systems that can dynamically adapt their underlying mechanisms according to the characteristics of the program. DYNTM records statistics of previously executed transactions to select the best-suited strategy each time a new instance of a transaction starts. SWAPTM takes a different approach: it tracks information of the current transactional instance to change its priority level at runtime. Both alternatives obtain great performance over existing proposals that employ fixed transactional policies, especially in applications with phase changes.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The emerging shift toward Chip Multiprocessor (CMP) architectures [28, 57, 62] moved the pressure of how to exploit effectively hardware resources to the programmer [87]. While in the past software developers were able to transparently speed up sequential applications using processors with ever higher instruction-level parallelism [127], nowadays they are forced to write multithreaded applications in order to extract thread-level parallelism. Thus, the success of future generation CMP processors highly depends on the capacity to widely adopt parallel computing as a standard programming paradigm.

Most parallel programming models have relied during decades on blocking synchronization for mutual exclusion of critical sections in multithreaded applications (*i.e.*, chunks of instructions that atomically modify shared memory variables), although it is well-known that protecting critical sections with *locks*—atomic variables that are acquired (released) before (after) executing the critical section—is error-prone, it lacks composability and may limit overall performance [29].

So far, two distinct strategies have been followed when developing parallel software using blocking synchronization. Easy-to-use *coarse-grain* locking protects a large amount of code with the same lock, which in turn serializes critical sections and, therefore, limits application's scalability. Difficult-to-develop *fine-grain* locking employs small critical sections to increase the performance of parallel programs, but tuning an application is extremely hard and may lead to programming errors (*e.g.*, deadlocks) [118]. This difficult to address trade-off between high performance and ease of use has encouraged research on alternative, lock-free parallel programming models [53].

This thesis deals with how to incorporate hardware support in modern CMP environments for Transactional Memory [49], possibly the most promising (and broadly accepted) non-blocking parallel programming approach. To this end, in this thesis we try to show how programmer-friendly multithreaded applications can achieve good performance when the system devotes part of its transistor budget to enhance concurrency.

1.1 Transactional Memory

Transactional Memory (TM) provides a parallel paradigm that facilitates application writing without sacrificing performance [102]. In TM, the programmer decomposes the application in different threads that are executed simultaneously, and then encapsulates blocks of code that access shared memory inside *transactions*. Like in database systems, the underlying TM mechanisms guarantee that transactions follow the **ACI** properties [41]: **A**tomicity (transactions are either fully completed or not executed at all), **C**onsistency (completed transactions can be deterministically ordered) and **I**solation (transactions must not report data races with concurrent threads during their execution).

Efficient TM systems employ speculation in order to execute transactions concurrently. On the one hand, this technique allows the system to get rid of deadlocks (all transactions are eventually executed) and enables the composition of atomic blocks because deadlocking is no longer possible. On the other hand, *optimistic* TM execution permits the system to run simultaneously transactions from independent threads, while *pessimistic* locking execution serializes critical sections, even when they do not access the same data. In consequence, TM systems commonly perform (and scale) better than systems that utilize hand-coded locks, especially when they execute many-threaded or coarse-grained applications.

In contrast to blocking synchronization schemes, in TM is the system, not the programmer, who ensures the correct execution of the program. Programming languages designed on top of a TM system just need to provide semantics to define *where* a transaction starts and finishes,

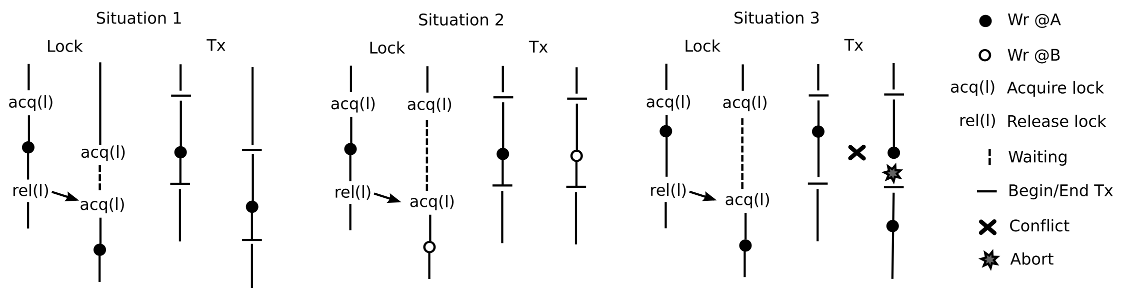


Figure 1.1: Lock- and transactional-based multithreaded executions

hiding from the user *how* the transaction is executed. Thus, all synchronization events are carried out implicitly, which simplifies significantly the development of parallel software.

In order to provide safe and efficient transactional execution, the system must implement a set of invisible mechanisms to verify at runtime that the ACI properties are not violated. Therefore, before *committing* a transaction (*i.e.*, finishing speculative execution and making transactional state *globally* visible) the system must check that no *conflicts* (*i.e.*, memory access that breaks the isolation invariant) involving this transaction exist. However, if a violation is reported, the system may potentially *abort* (*i.e.*, undo all the memory modifications performed inside the transaction) one of the conflicting transactions and restart it from its very beginning.

Figure 1.1 shows the differences between lock-based and transactional-based executions. While locking techniques block the execution of critical sections when running a piece of code protected by the same lock, the transactional (speculative) strategy permits non-conflicting transactions to execute—and even commit—in parallel (Situation 1 and 2). Nonetheless, Situation 3 generates a conflict between two transactions, which produces the abort of the requesting transaction. Notice that this scenario does not hurt concurrency compared to the lock-based execution, given that, in the worst case, the aborted transaction will restart at the time that the conflict disappears—this happens when the conflicting transaction commits, which approximately coincides with the time that the lock is released under blocking synchronization.

Environments that sustain TM can be implemented exclusively in software (Software Transactional Memory [48, 77, 104, 110], STM for short), mostly in hardware (Hardware Transactional Memory [45, 49, 84, 97], HTM for short) or a combination of the two (Hybrid Transactional Memory [31, 63, 65], HyTM for short). Complete specifications of TM systems can be found in [46], where Harris *et al.* cover in detail the large TM design space up to early 2010.

The majority of STM systems use software locks to *locally* hold the ownership of data objects, and record the memory locations accessed within transactions in software structures, which must be constantly traversed during transactional execution. Hence, STM systems incur considerable delays when running transactional code. In fact, some researchers claim that STM is just a testing tool to familiarize users with the TM programming model [18]. Instead, HTM systems devote silicon to accelerate transactions, resulting in significantly less overheads than STM systems, although they lose the portability to legacy (non-HTM) systems. Moreover, HTM systems are more robust, preserving strong isolation between transactional and non-transactional code [17].

HyTM systems propose an interesting compromise between *high-performing* hardware and *flexible* software [31]. As in any compromise, it is difficult to know exactly where the optimal software-hardware division lays. Moreover, any fully-hardware TM system requires software to enable virtualization of large transactions. As a consequence, most HTM systems can somehow be considered hybrid approaches.

Among all the flavors of TM proposed by today, we think that HTM is the one that offers the best alternative, especially in terms of performance. We believe that, if the TM programming model becomes ubiquitous, it will be through some HTM implementation [33].

1.2 HTM Systems: Problems and Limitations

To the best of our knowledge, almost all proposed HTM designs clearly fall into one of the two possible categories: they can be either *eager* or *lazy*. Each category defines a set of actions that must be taken when the system has to resolve memory inconsistencies introduced by in-flight transactions. This mechanism, commonly known as *conflict management* (CM), also establishes the strategy that the system follows when dealing with the speculative state, a group of rules commonly known as *version management* (VM).

Existing HTM systems fix the CM and VM policies at design time. The qualitative and quantitative analysis performed in this thesis shows that inflexible HTM systems are faced with, at least, one of the following limitations that discourage their implementation: (a) they ***perform poorly*** when executing non-trivial transactional workloads, (b) they ***do not scale*** on many-core systems, (c) their ***hardware*** cost is ***too expensive*** or (d) the complexity of the system makes its ***implementation not affordable***.

Eager HTM systems [5, 49, 84, 97]—those that resolve conflicts as soon as they happen—present a major variation on their design: they use opposite strategies to manage data versioning. *Late* (also known as lazy) VM systems rapidly recover the transactional state in hardware—it is kept in local caches—but utilize either software or very complex (and inefficient) hardware implementations to maintain overflowing data (*i.e.*, memory lines evicted from local caches), while *early* (also known as eager) VM systems employ slow software to restore the pre-transactional state, but handle overflows gracefully by storing in-place speculative data values. Because of this both approaches may suffer important performance penalties when executing applications with large transactions—early VM systems spend a lot of time recovering aborts, while late VM systems slow down transactional execution due to resource overflow.

Lazy HTM systems [20, 45, 112]—those that resolve conflicts at the time a transaction attempts to commit—are forced to implement *late* data versioning; therefore they are also affected by the slow-on-overflow issue. As the majority of lazy HTM designs introduce specialized (and cumbersome) structures to buffer the overflowing speculative state, they increment the hardware cost when compared to eager HTM systems. Moreover, lazy HTM systems require arbitration and communication with shared resources at commit time [22, 92, 124], which (i) substantially delays transactional execution, especially in applications that frequently commit transactions, (ii) increases the complexity of critical system elements, such as the directory or the coherence protocol, and (iii) hurts the system scalability.

Most importantly, fixed-policy (either eager or lazy) HTM systems establish the CM strategy at design time, taking always the same decision to resolve conflicts for the whole execution. It is widely known that eager and lazy HTM systems have their strengths and weaknesses [14], but, unfortunately, they are too inflexible in the way they manage transactional contention, resulting in a significant performance opportunity loss when they deal with complex transactional workloads that combine transactions of different size and variable contention [16].

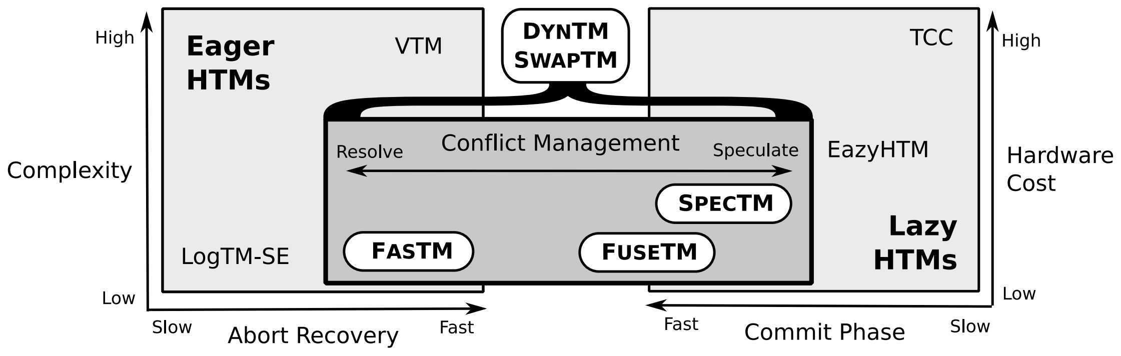


Figure 1.2: Intrinsic properties of the HTM systems proposed in this thesis

1.3 Thesis Contributions

All the above has led us to conceive **five** different ***high-performance*** HTM systems based on efficient eager and lazy HTM approaches. What is more, our proposals enable the ***scalability*** of coarse-grained TM applications, while keeping the ***design simple*** (system complexity remains under affordable limits) and requiring ***little hardware cost***.

This thesis tackles the major problems across the HTM design space. Our first contribution, FASTM, presents a novel solution for managing data versioning on eager HTM systems, whereas our second proposals, FUSETM and SPECTM, improve on prior lazy HTM works by including lightweight hardware to support multiple versions of the speculative state in a typical cache coherence protocol. Our last contributions, DYNTM and SWAPTM, integrate opposite conflict management policies in a single framework to select the most profitable strategy according to the application characteristics. Figure 1.2 presents a pictorial overview of the HTM systems presented on this thesis and compares them with well-known, state-of-the-art HTM implementations.

1.3.1 FASTM

The data VM mechanism is possibly the main limitation factor of eager HTM systems when executing large transactions. Late VM systems (*e.g.*, VTM [97]) suffer important overheads on commits and on resource overflows, while early VM systems (*e.g.*, LogTM-SE [130]) experiment long delays in case of abort, given that the old state is recovered using software. Looking at the available HTM systems, one could conclude that the shortcomings of each VM style are inherent to its design philosophy, which defines *how* (software or hardware) and *where* (in private or in shared resources) the speculative/non-speculative state is kept.

Our first contribution, FASTM (log-based HTM with fast abort recovery), is a departure from this thinking by observing that the important thing for VM is *who* the owner of the speculative state is and *when* the state should become visible to the rest of the system, but not *where* or *how* the (non-)speculative state is stored. Thus, FASTM breaks with the VM implementation dichotomy and proposes a change in the design philosophy: a *hybrid* VM alternative that takes advantage of the strong points of both approaches to **accelerate commits and aborts** while implementing a **simple overflow policy**—in fact, FASTM can be seen as an early VM system with a late VM implementation. These VM properties become a requirement for complex TM workloads, which are believed to represent future transactional applications.

1.3.2 FUSETM and SPECTM

Early work on lazy HTM systems propose *ad hoc* mechanisms to implement a coherence and consistency model based on transactions (TCC [45] for short). These designs introduce some modifications (few of them non-trivial) on different layers of standard CMP configuration (memory hierarchy, coherence protocols, on-chip directory, *etc.*) that increase the complexity of the design. Although recent proposals have tried to generalize the TCC approach to integrate it in a traditional CMP environment [22, 92], most solutions still require extra communication on commits and specialized hardware to buffer non-validated data.

Our second contribution seeks to decouple data versioning from conflict management by adapting transactional mechanisms under conventional (eager-like) hardware. We show how this goal can be achieved through the implementation of two distinct HTM systems. FUSETM introduces a *flexible* and *simple* VM framework to track (and defer if necessary) memory violations from lazy-mode transactions without adding unaffordable complexity in the system. Moreover, by extending its VM strategy to enable multiple versions of the same data and deferring directory updates, the system is able to **remove data transfers and communications at commit time**, which is very useful in applications with short transactions. FUSETM (fused HTM system with local commits) also offers **an eager mode of execution for those transac-**

tions that exceed private resources, so the system does not have to provide complex hardware support for boundless lazy transactions.

FUSETM falls back to "on-demand" resolution of conflicts for overflowing transactions, which may restrict overall concurrency. SPECTM (speculative HTM system with early overflowing updates) tries to overcome this limitation by offering a two-level data versioning mechanism: multiple copies of a line are allowed in the first level caches, whereas a single copy is permitted after evicting the line toward the upper levels of the memory hierarchy. This approach **enables deferred conflict management for any kind of transaction**—even for those that exceed private buffers.

1.3.3 DYNTM and SWAPTM

Prior HTM systems fix the conflict management policy at design time. Fixed-policy HTM systems are faced with numerous issues that limit the concurrency of transactional applications. Experiments presented on this thesis show that the two groups (eager or lazy) of HTM systems do not respond equally to all types of workloads, which is crucial given the unknowns about the behavior of future TM applications. A truly flexible HTM that could select the ideal execution mode for each transaction at runtime would be more adept at dealing with many different types of workloads.

Our last contribution pursues the design of such a fully-flexible HTM system. More specifically, we propose two such systems. DYNTM (dynamically adaptable HTM system) combines eager and lazy transactions simultaneously to **choose the best performing mode of execution** for each dynamic instance of a transaction. DYNTM uses a simple (and local) predictor to dynamically decide at the beginning of a transaction the best-suited (eager or lazy) execution mode. The election, which is hidden from the programmer, is based on the behavior of *past* instances of the same transaction. This system greatly outperforms fixed-policy HTM systems [112].

Once the philosophical barrier of eager versus lazy HTM systems is crossed, a whole new class of opportunities for research is opened—DYNTM is only the first implementation. One interesting optimization is SWAPTM (high-performing HTM system with swapping execution modes), a dynamic alternative that **switches the transactional mode of execution of trans-**

actions on the fly. SWAPTM offers early VM for unbounded lazy transactions, thereby the transactional execution mode is not restricted by the size of the transaction. SWAPTM analyzes the characteristics of each *individual* instance of a transaction to decide its performance impact, and then adjusts the underlying hardware to select the most adequate system configuration.

1.4 Relationship to My Previously Published Work

FASTM [71] was published in the *Proceedings of the 18th International Conference on Parallel Architectures and Compilers Techniques* (PACT'09), along with co-authors Grigorios Magklis and Antonio Gonzalez. FASTM was motivated by a potential study that had appeared in Lupon's Master Thesis [69] and by the limitations found while evaluating a log-based store-buffered HTM system [70], a study that was published in the *Proceedings of the 9th Workshop on MEmory Performance: DEaling with Applications, systems and architectures* (MEDEA'08).

This thesis extends earlier published work by proposing new variations in the design and evaluating the system with a wider spectrum of benchmarks. A light description of the selective logging and wake-up notification mechanisms for FASTM can be found in a Technical Report [75]. This thesis complements previous work with a detailed discussion of hardware alternatives that permit virtual address logging.

The FUSETM system was published in the *Proceedings of the 43rd International Symposium on Microarchitecture* (MICRO'10) under the label of the *lazy* execution mode for DYNTM [72]. This thesis describes the FUSETM system in a greater detail and provides a more exhaustive characterization. The SPECTM system together with the hardware support for unbounded lazy transactions is described in a Technical Report [74].

Respect to DYNTM [72], this thesis presents a more accurate discussion over related work and describes from top to bottom the implementation of the conflict management policy and the configuration of the transactional mode selector. The work on SWAPTM is currently under submission. An early version of that creation is described in a Technical Report [73].

1.5 Thesis Organization

This thesis dissertation is organized as follows. Chapter 2 reviews the state-of-the-art of TM systems, paying more attention to those that include hardware support. The chapter starts given an historical overview of TM proposals and follows discussing the mechanisms that forge modern HTM proposals and how they are implemented. Then, it reviews how these mechanisms are used to build fixed-policy (either eager or lazy) HTM systems, pointing out the main limitations of prior work.

Chapter 3 presents the experimental methodology followed through this thesis. It begins explaining how the simulation infrastructure models the processor, the memory hierarchy and the transactional hardware support. Then, the chapter defines the base CMP configuration and the system parameters utilized along the evaluation, together with the reference HTM systems implemented as baselines. After that, it exposes the TM benchmark suites used to evaluate the correctness and the performance of the proposed HTM systems, classifying them into different categories. Finally, the chapter ends explaining the experimental methods and metrics adopted in the evaluation.

The next three chapters describe the contributions of our research. All three chapters briefly introduce the work with a motivation section, and present a big picture of the contribution. Then, they explain the intrinsic details of the proposals (hardware support, memory operations, transactional mechanisms and so on) and some design optimizations. From that point, each chapter evaluates the proposed implementations and compares them with baseline HTM architectures. After that, a qualitative comparison against related work is performed, concluding with a summary of the exposed ideas and results.

Chapter 4 presents FASTM as a revolutionary eager HTM system with a novel data version management mechanism, and extends the proposal with two additional implementation variants. Chapter 5 describes FUSETM and SPECTM as pure, not-so-complex lazy HTM designs. It also analyzes the benefits and drawbacks of using speculative conflict management and compares FUSETM's and SPECTM's performance against FASTM. Chapter 6 takes the results of the prior chapter to motivate the evolution to two unified, truly flexible and adaptive HTM systems (DYNTM and SWAPTM). After detailing the innermost parts of both alternatives, this chapter studies how transactional applications behave under documented HTM systems to show the importance of implementing high-performing designs.

Chapter 7 concludes the research presented on this thesis and avenues for future work.

Chapter 2

Background on Transactional Memory

In the late 70's, Lomet came up with the idea of using database transactions when accessing shared data [68]. It was not until two decades ago though, when Herlihy and Moss introduced the concept of Transactional Memory (TM) as a new programming paradigm that intended to make lock-free mechanisms more efficient than blocking synchronization techniques [49]. Since then, many researchers have taken different approaches to construct efficient TM systems. This section starts describing those alternatives, and follows presenting the mechanisms underneath hardware-assisted TM (HTM) systems. Afterwards, it reviews most notable HTM (fully-hardware, hardware-accelerated and hybrid TM) implementations and discusses their complexity, performance and hardware cost.

2.1 Transactional Memory Systems

A large amount of environments combine different degrees of hardware and software support to execute speculative transactions. Harris (second edition [46]) joined Larus and Rajwar (first edition [64]) to synthesize a computer architecture lecture that offers an extensive survey of the state of the art on TM systems, as of early spring 2010. An updated TM bibliography can be found in the University of Wisconsin website [1].

We draw some impressions about most commonly used transactional systems in Figure 2.1, which shows a graphical representation of some TM implementations and their basic properties.

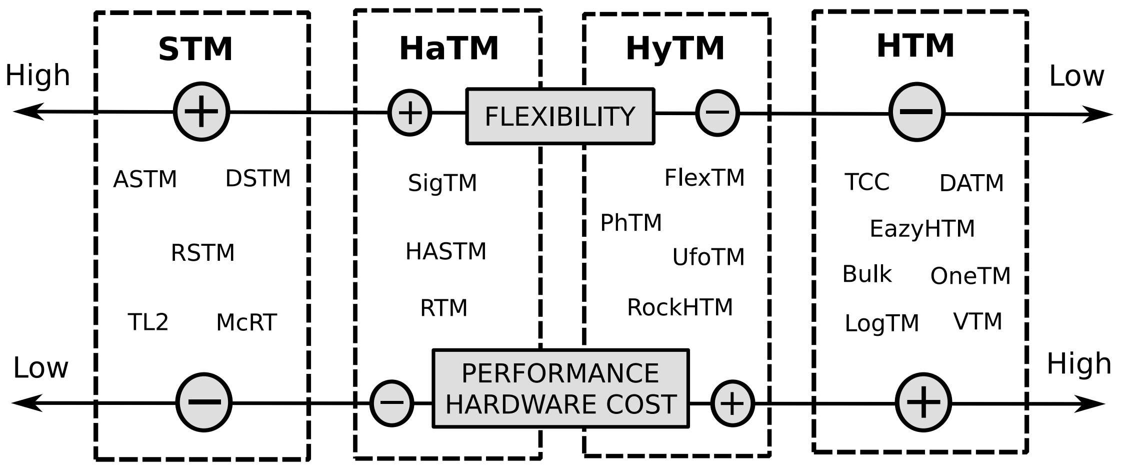


Figure 2.1: Implementations and properties of STM and HTM systems

As it can be seen, software strategies are cheaper and more flexible, but achieve poor performance compared to hardware-assisted TM systems. The following subsections highlight the main features of software- and hardware-based TM approaches.

2.1.1 Software Transactional Memory Systems

Shavit *et al.* proposed Software Transactional Memory (STM) as a friendly interface to execute transactional applications in mainframe systems [110]. In STM, memory accesses within transactions have to access a software library that implements automatic, object-grained locking and track version numbers using data structures. In order to keep the system consistent, software libraries must synchronize transactional reads or updates upon those structures, limiting the concurrency of the parallel execution (see Cascaval [18] *et al.* for more details).

STM systems provide high flexibility and can easily be revised. Moreover, they (i) can conveniently handle transactions of any size or duration, (ii) require simple validation and (iii) can run on legacy hardware—which makes them serviceable for the development of transactional programs. Nonetheless, software monitoring sacrifices performance and power, as it requires explicit (programmed by hand) calls to system libraries each time a memory location is accessed, which results in the execution of additional instructions. In order to overcome this vital constraint, several STM systems have been proposed.

Dynamic STM (DSTM [48]) is a deferred update STM system implemented as a library usable in C++ or Java. It introduces a flexible conflict manager that delegates to the programmer how conflicts are resolved, uses obstruction freedom as a non-blocking progress condition and permits early object release, which reduces the size of the read set before committing. Word-granularity STM (WSTM [78]) performs similar to DSTM, but detects conflicts with a highly accurate precision. Scherer and Scott [108] studied the behavior of different conflict resolution policies for the DSTM system and concluded that no policy performs best in all the measured scenarios. To optimize the base DSTM contention manager, Adaptive STM (ASTM [109]) changes the conflict resolution policy at run-time.

Rochester STM (RSTM [77]) enhances the performance of deferred update STM systems by reducing the levels of indirections to an object. It also provides its own memory allocator and uses reader invalidation on commits, which substantially minimizes the size of those data structures devoted to maintain the read set. InvalSTM [40] complements RSTM's approach by providing full invalidation, which accelerates transactions with big read and/or write sets in many-threaded applications.

In contrast to the above proposals, Transactional Locking (*e.g.*, TL2 [34]) STM systems combine deferred state updates with blocking synchronization to hold non-committed values. When a transaction finishes its speculative execution, it must lock the modified objects—*i.e.*, acquire a conventional lock for each element of the write set—using a global clock, what allows the transaction to validate its read set. The benefit of locking objects consists on simplifying software data structures and minimizing the transactional overhead.

In the manner of TL2, McRT-STM [104] and Bartok STM (BSTM [47]) use transactional locking for keeping the consistency of their read and write sets. Unlike TL2, McRT-STM and BSTM perform direct update, maintaining transactional values in-place in memory and using early blocking synchronization—*i.e.*, acquiring locks at the first time an object is accessed—to prevent other transactions to read or write the modified state. Furthermore, they combine pessimistic concurrency control for updates with optimistic control for reads, and use version numbers on a per-object basis instead of a global clock. Those implementations can take advantage of Intel's C++ STM [4] and Microsoft's Bartok [3] compilers to aggressively reduce the size of software data structures.

TinySTM [37] is a lightweight STM system written in C that borrows several key aspects of TL2. It implements word granularity and uses a timestamp algorithm based on LSA [100] to resolve conflicts. SwissTM [35] mixes TL2-like global clocking with a hybrid conflict detection mechanism. RingSTM [114] and STMlite [83] implement software Bloom filters to avoid storing recurrent metadata in the heap space.

Each of the above optimized STM implementations intend to leverage the overhead associated with software transactional mechanisms [110]. Nevertheless, the question of how to build an efficient STM remains open. Recent studies show that STM systems underperform lock-based executions—especially when few threads are used [125]—leading some academics to postulate that STM is just a mere research toy [18]). For this reason, it seems inevitable to conclude that some kind of hardware support is necessary in order to speed up transactional execution. Next subsection overviews and classifies those TM systems that incorporate hardware assistance in some (or all) of their layers.

2.1.2 Hardware Transactional Memory Systems

Herlihy and Moss included hardware support in the microarchitecture of their original TM design, building a hardware-assisted TM (HTM) system [49]. Their approach uses typical cache management and coherence protocols on non-transactional operations, and provides a new Instruction Set Architecture (ISA) for transactional accesses, commit actions and state validation. A separate processor cache contains old and transactional values, which can only be accessed by the owner processor.

Hardware-assisted TM systems are less invasive than STM systems, given that they treat all memory accesses within a transaction as implicitly transactional. Thus, they only demand two additional instructions in the architecture to encapsulate blocks of atomic instructions inside transactions: `Tx_Begin` and `Tx_End`. This way, memory operations performed inside those bounds can transparently use built-in transactional hardware with little (in some cases almost negligible) performance and power penalty.

Proposals for HTM implementations have been around for more than a decade, feeding a wide range of design possibilities. Fully-hardware TM environments (commonly generalized as HTM) fill the system with specialized mechanisms in order to accelerate whatever transaction, reducing the overheads produced by special operations to the bare minimum. Actually, some HTM designs introduce no overhead for standard transactional execution. However, they still require some kind of software support for virtualization purposes. LogTM [84], TCC [45] or VTM [97] are few examples of fully-hardware HTM implementations.

Hybrid TM systems (HyTM for short) provide finite hardware support, devoting best-effort transistors for conventional transactions and relying on STM systems for those transactions that do not fit in the hardware or require unusual actions. Early HyTM approaches [31, 63], PhTM [65], FlexTM [112] or UFO [8] are instances of partially-in-hardware HyTM designs. Hardware-accelerated (HaTM) systems share some similarities with hybrid TM systems, as their execution depends upon a STM. However, rather than falling back to software in corner situations, HaTM systems always run on a faster STM mode that uses small pieces of acute hardware to speed up the slow software infrastructure. HASTM [105], RSTM [113] or SigTM [17] are well-known representations of HaTM systems.

The next section explains in detail the main mechanisms used to build high-performance fully-hardware HTM, HyTM and HaTM systems. Given that the line to distinguish the category of each approach is extremely tight, we decided to refer all of those systems as HTM[1], describing their implementation components in Sections 2.3 and 2.4.

2.2 Hardware Transactional Mechanisms

Hardware transactional mechanisms are necessary in order to track memory locations read or written inside the transaction (*access summary*), buffer both the previous (old) and the speculative (new) memory state and restore the old values in case of abort (*version management*), and detect and resolve conflicts among transactions (*conflict management*).

Hill *et al.* [51] proposed a decoupled implementation of transactional mechanisms. Decomposing hardware into interchangeable components aids HTM design and permits the system to use those mechanisms for other purposes, such as reliability, security, deterministic replay and high-performance sequential (even parallel) execution. To this end, a TM system must provide an efficient implementation of these mechanisms, offering fast execution in case of infre-

[1]After all, any HTM system introduces specific hardware mechanisms and requires software solutions to handle memory paging or context switches.

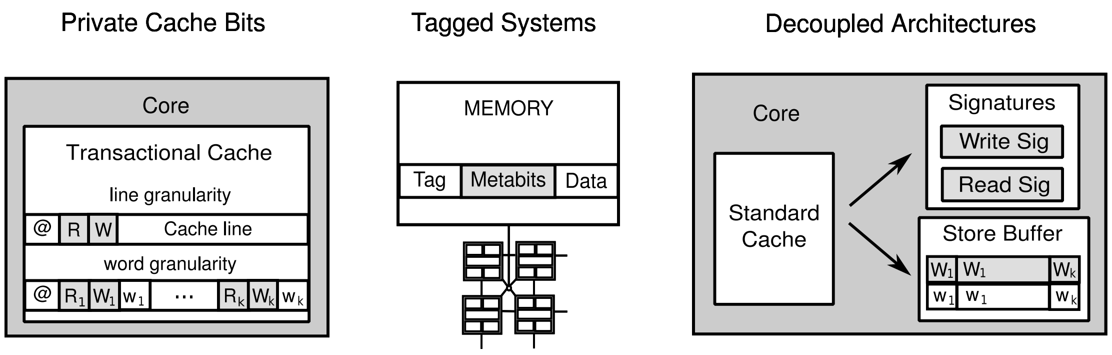


Figure 2.2: Hardware implementations of the acccess summary mechanism

quent conflicts and minimizing the impact of collisions among transactions when contention is present. In this section, we explain which is the purpose of hardware transactional mechanisms and overview their most-known design options.

2.2.1 Access Summary

Access summary is the mechanism that tracks the set of memory locations accessed by a transaction, commonly known as read (for transactional loads) and write (for transactional stores) sets. A memory address is inserted in either the read or the write set—depending on its access type—when it finalizes successfully. It is necessary to maintain those memory accesses in silicon to rapidly match the addresses that induce conflicts among transactions. Several implementations of the access summary mechanism have been proposed in the literature. Figure 2.2 provides a schematic view of some of them.

Early HTM proposals introduce read and write (R/W) bits in private caches (typically in the L1 cache [45, 84, 97]) that have to be asserted when a memory operation completes and tested each time a remote memory request is forwarded to a processing core. In some cases, these bits are coupled with an additional bit that informs if a transactional line has been evicted from a cache set [5, 25]. While most of the HTM proposals track the conflict at the granularity of a line, optimized systems may introduce R/W bits per word with a noticeable increment on cache area [81, 85]. In order to simplify hardware logic, R/W cache bits can be replaced with supplementary cache states, requiring fewer bits and integrating transactional actions in the cache coherence protocol [113].

Previous mechanisms are limited by the size of private caches, losing precision when the system replaces a cache line. In those occasions, systems may behave as HyTMs, requiring software [31] or OS support [24] for tracking evicted lines. Another design alternative consists on implementing a tagged (also known as supervised) memory, where some metabits are kept in physical memory for a variety of purposes including data race detection, determinism control or typestate trackers. Transactional Memory can also take advantage of this system configuration when metadata is used to hold transactional read and write sets [8, 12]. Although supervised programs may experiment ordering issues when they are executed under a weak consistency model, recent research precludes that this problem does not affect TM systems [13].

Even though R/W bits offers a low-complexity solution for recording memory accesses, its implementation is not-so-attractive from commercial point of view. As industry always tries to keep cache design simple and unmodified, they would prefer not to include transactional support in caches and rely instead on decoupled solutions. To this end, some researchers investigate the use of signatures as an interesting alternative to eliminate transactional state from caches [20, 106, 130].

A signature consists on a Bloom filter [9], where a set of memory addresses are collected in an array of bits. Each time that a memory operation is retired, the system must insert its address in the signature by encoding the given address with a hash function and then marking certain bits of the array. Testing operations are performed similarly, applying the hash function over remote addresses and checking if all the selected bits are asserted. These basic operations can be extended with join and intersection functionalities, which are required in Bulk implementations [21, 93].

Ceze *et al.* introduced signatures in order to improve the capacity offered by R/W bits [20]. However, signatures may saturate when they contain lots of addresses, losing precision and thus creating *false positives*—*i.e.*, systems detect that an address is present in a read or write set, while it is not. Lots of works studied the effect of transactions in signatures, and they showed that true Bloom filters—those that can mark any bit of the array—coupled with simple bit-selection hash algorithms generate lots of false conflicts, what may hurt overall performance.

Thus, several signature implementations have been proposed to reduce the rate of false positives. Sanchez *et al.* presented parallel and Cuckoo Bloom filters as more efficient organizations

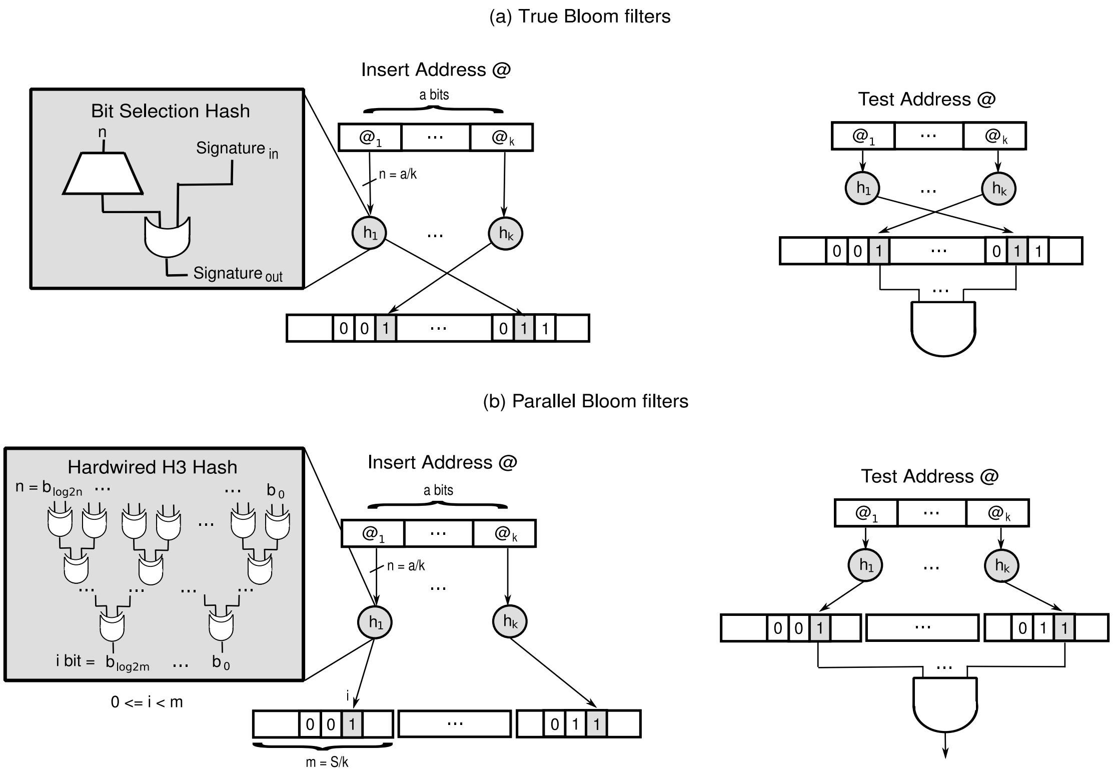


Figure 2.3: Design options when implementing signatures

for signatures [106]. Parallel filters use a finite number of hash functions that point to smaller array blocks to reduce the area occupied by signatures. Cuckoo-Bloom filters increase even more the accuracy of signatures by providing an exact representation of their content when the number of items is low and by behaving like a parallel filter after bypassing a saturating threshold.

Yen *et al.* studied the costs of implementing bit-selection, H3 XOR and page-block XOR hash functions [131], concluding that XOR-based hash functions help to maximize the precision on signatures. Quislant *et al.* demonstrated the importance of using space locality when inserting addresses in signatures [94]. Figure 2.3 compares the insert and test operations in true (bit-selection hash function) and parallel (H3 hash function) Bloom filters.

Despite signatures, there are simpler decoupled access summary mechanisms. For instance, small fully-associative structures (*e.g.*, Store Buffers) can be used to track modest write sets [33, 49]. However, this mechanism restricts the number of writes enclosed inside a transaction.

2.2.2 Data Version Management

Data version management (VM) is one of the key design dimensions of a TM system, as its implementation impacts directly on the performance and the complexity of the system. Version management defines how and where transactional modifications are stored, and what actions must be performed at commit and abort time. The majority of TM systems fall into one of two distinct strategies for version management: **early** (also known as *eager* or *in-place*) or **late** (also known as *lazy* or *deferred update*).

On the one hand, late version management [5, 24, 45, 49, 97] keeps old (pre-transactional) state in-place in memory and buffers new state (values generated inside the running transaction) elsewhere. This makes aborts fast, but commits have an overhead because the new state must become globally visible. Thus, data movement is necessary in order to update shared components such as upper levels of the memory hierarchy or the directory.

Most late VM systems use the L1 caches to buffer new state, and specialized coherency protocols to hide transactional updates from the rest of the memory hierarchy. LV* [86] is just the *Nth* HTM system that implements late version management. Other implementations, like the one proposed for the Rock [79] processor, store transactional modifications in a gated store buffer, the content of which is drained at commit time.

In case the new state overflows its buffering space, some late VM systems behave similar to HyTM systems, and fall-back to a STM implementation [110]. Some other HTM systems [5,97] store overflowed state in a data structure kept in memory, which must be accessed on cache misses and on commits. Falling-back to STM incurs into significant performance loss while fully-hardware HTM systems require complex, expensive and cumbersome hardware mechanisms. This fact makes transactional state overflows the main drawback of late VM systems.

On the other hand, early version management [10,12] puts new state in-place in memory and buffers pre-transactional state elsewhere, usually a software-managed log structure in cacheable memory [84]. This makes commits fast, since data is already stored in memory, but aborts have an overhead because the old state must be recovered.

Also, since the pre-transactional state is stored in the log and can be recovered, transactional modifications can be put anywhere in the memory hierarchy, so early VM systems do not suffer

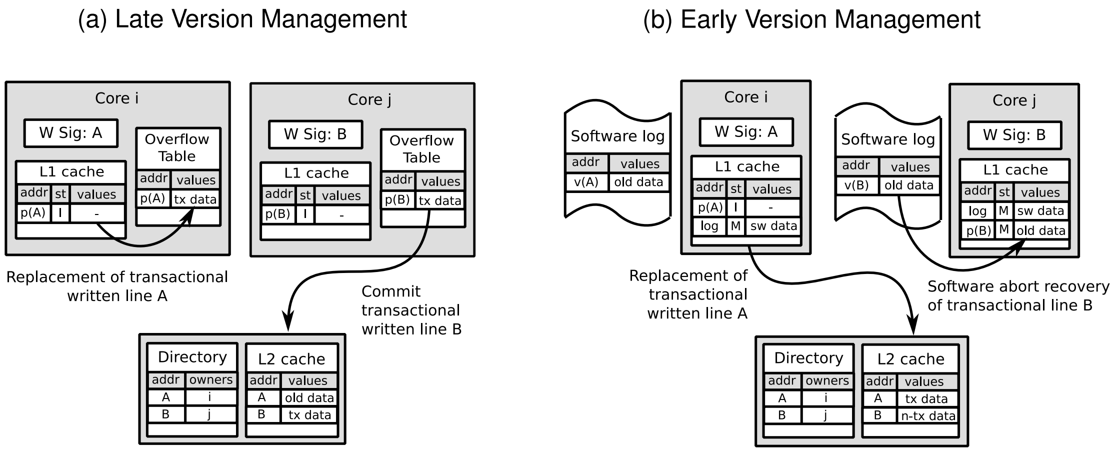


Figure 2.4: Data version management alternatives in HTM systems

from cache (or store buffer) overflows like those systems with late VM, with the additional benefit of reduced hardware cost. LogTM-SE [130] is an example of an early VM system.

Figure 2.4 schemes how late and early version management HTM systems operate. In Figure 2.4a, cores `Ci` and `Cj` introduce a specialized on-core hardware structure to manage transactionally written evicted data, keeping pre-transactional data in the L2 cache. When `Cj` commits, it must traverse this structure and atomically update the shared L2 cache data with the new, not-anymore-speculative data. Instead, Figure 2.4b shows how early VM cores (`Ci` and `Cj`) gracefully handle cache overflows by moving evicted data to the L2 cache. However, core `Cj` requires a slow software routine to recover the pre-transactional state, which is stored in a private, cacheable and software-accessed log.

2.2.3 Conflict Management

Conflict management (CM) is possibly the most critical aspect of HTM systems. It is the hardware mechanism in charge of preserving the transactional isolation property and the correct ordering of the committed transactions. Furthermore, it is also necessary to guarantee forward progress in case of a memory race. First, it has to determine how and when conflicts are detected. This feature is commonly known as conflict detection. Second, it must decide when and through which technique conflicts are resolved. The last contribution of the mechanisms is defining which actions are performed (and by whom) in order to eliminate the collision that originated the conflict. The last two exposed facets are sometimes referred as the conflict resolution policy.

Conflict management policies are classified as **eager** and **lazy**. Eager CM schemes detect conflicts at the moment that a load (store) instruction from an in-flight transaction accesses a memory location being written (read or written) by another in-flight transaction [98, 130]. The majority of eager implementations slightly modify the coherence protocol to introduce an implicit test operation against the read (for writes) or write (for writes and reads) sets from the owners of the requested line [5, 49, 97]. For instance, LogTM [84] uses sticky states to hold the last owner (or sharers) of an evicted line and enforce consistency checks even if that processor does not maintain the line anymore.

Eager CM strategies normally cohabit with conservative conflict resolution policies—those that resolve memory violations at the time that they are produced. These policies disallow inconsistencies among transactions and permit the system to keep a single version of a written line, making the data versioning mechanism straightforward. Nevertheless, eager conflict resolution policies are less flexible in the way they manage contention and may generate performance pathologies such as unfair scheduling of transactions [14] or multiple crossed aborts [111].

Primitive HTM systems implemented a requester-wins policy that served pre-transactional values to those processors that request a transactional line, but abort the owners of the line to avoid discrepancies. As this strategy creates repeated aborts that may produce livelocks, some HTM systems introduce a software backoff to spread contention. Advanced HTM implementations stall requesting petitions—*i.e.,* retry the memory operation until it succeeds—once they detect a conflict, using a timestamp-based protocol to eliminate dependence cycles between stalled transactions [96]. This approach enables the conservation of non-conflicting, consistent transactional work.

In contrast, lazy CM schemes resolve conflicts at the time that a transaction attempts to commit [22, 45, 92]. Before making the transactional state visible, processors must abort all the transactions that have accessed the committing write set, imposing a consistent order between transactional instances. Notice that transactions are only aborted when another transaction wants to commit, so progress is not an issue when applying lazy policies.

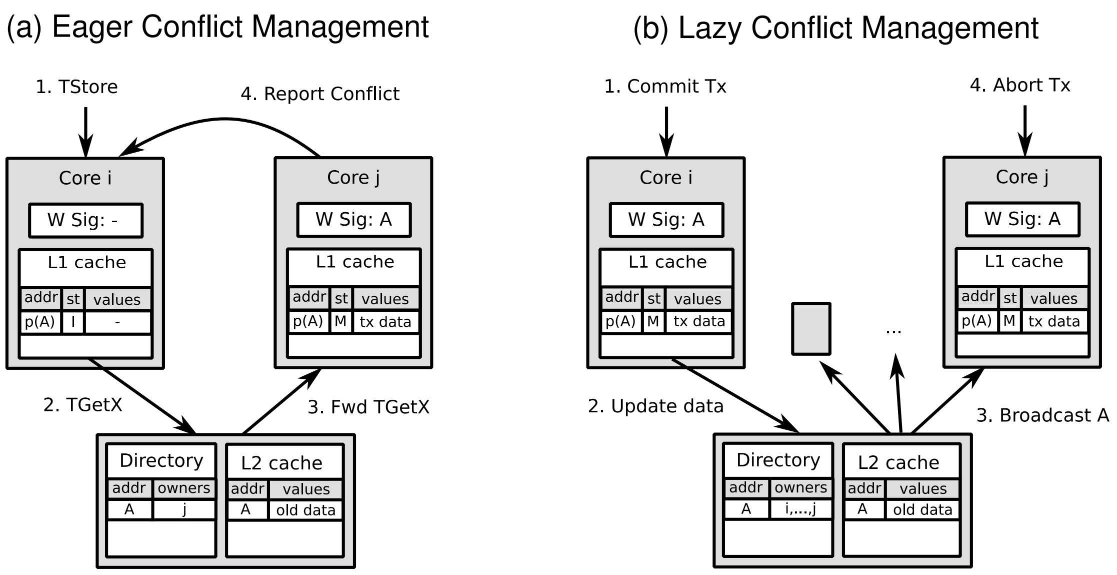


Figure 2.5: Conflict management alternatives in HTM systems

In lazy schemes, conflict detection can take place early [112, 124] or it can be delayed until commit [82, 93]—after all, the conflict will not be resolved until commit time. Early conflict detection proposals integrate sanity checks in the coherence protocol and track individually conflicts in hidden registers. Deferred conflict detection requires the broadcast of the write set [20, 45], a power-hungry technique that do not scale with many-core CMPs, albeit it may reduce the latency of memory operations by eliminating coherence messages [21].

Lazy CM strategies enable more concurrency between conflicting transactions, which keep executing even in the case of collision. This policy permits the system to (i) omit read-write conflicts (committing readers always load pre-transactional data), (ii) increment its flexibility, (iii) pre-fetch useful data if the transaction requires a re-execution and (iv) eradicate software management of conflicts (*e.g.*, software backoff). However, lazy CM schemes require late VM with buffering support for multiple data versions of the same cache line, arbitration and extra communication with shared resources at commit time and *ad hoc* hardware implementations. What is more, optimistic treatment of conflicts may deliver a large amount of discarded work.

Figure 2.5 shows how HTM systems operate under eager and lazy CM strategies. In Figure 2.5a, core `Ci` attempts to eagerly write line A, which belongs to the write set of core `Cj`. The directory forwards the request to the owner of the line (`Cj`), which checks its write signature

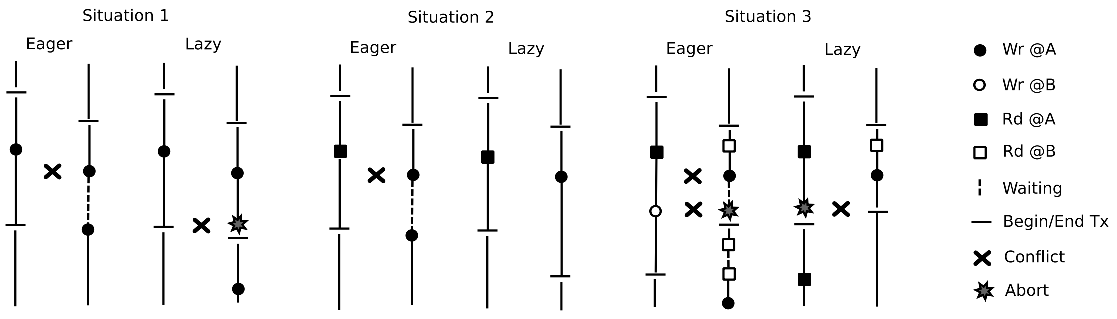


Figure 2.6: Eager versus lazy transactional execution

and replies `Ci` with a conflict message. Instead, in Figure 2.5b cores `Ci` and `Cj` have written line A within a transaction, which `Ci` attempts to commit. Before updating the directory and the L2 cache, the system broadcast the write set of `Ci`—in this case, line A—to abort inconsistent transactions. When core `Cj` receives the message, it automatically aborts its transaction.

2.2.4 Building High-Performance HTM Systems

Bobba *et al.* [14] pointed out that HTM systems reflect different choices while implementing the above mechanisms, dividing HTM implementations in two different groups: ***eager*** (*single* version per memory block, *immediate* resolution of conflicts) and ***lazy*** (*multiple* versions per memory block, *deferred* resolution of conflicts) HTM systems. Although hybrid approaches can be found in the literature, they can easily be placed in one of these two groups.

Figure 2.6 shows how eager and lazy HTM systems deal with conflicting transactional executions. In Situation 1, the eager HTM can preserve useful execution on write-write conflicts, while at least one transaction has to abort due to a direct inconsistency when it is executed in the lazy HTM environment. In Situation 2, the lazy HTM successfully speculates with the read-write conflict, while the eager HTM has to stall the conflicting request. In Situation 3, the eager HTM must abort a transaction when a potential cycle between stalled transactions is detected, while the lazy HTM may starve older transactions. In the following sections, we borrow the eager/lazy taxonomy to describe state-of-the-art HTM implementations.

2.3 Eager HTM Systems

We designate eager HTM systems those implementations that resolve conflicts as soon as they are produced, independently of the data version management strategy that they follow. Identifying conflicts once they are detected enables exercising conservative conflict resolution policies, which prevents the abort (and re-execution) of large transactions with one-direction conflicts. Moreover, eager HTM can employ *either* early or late version management, given that the system maintains just a *single* speculative value for any transactionally written data. In this section, we review related work in eager HTM systems. Table 2.1 summarizes the main characteristics of different-style eager HTM systems.

2.3.1 Bounded HTM Systems

Precursors of modern HTM systems observed that processors could use the coherence protocol to optimistically monitor atomic accesses on memory locations. Jensen *et al.* [59] were the first to propose synchronization primitives to track operations for a single memory address. Later, Herlihy and Moss [49] presented their pioneer HTM implementation, which supported finite-sized transactions. They added a specific transactional cache in the infrastructure to buffer written data explicitly classified as transactional. They also implemented an ownership-based coherence protocol to match incoming requests against the data stored in the transactional cache. Transactions kept running even in the case of conflict, but they required validation before committing. If someone had interfered with them during their execution, the register state had to be restored and transactional cache lines invalidated.

Speculative Lock Elision (SLE [95]) is an implicitly transactional implementation—it keeps using a lock-based nomenclature—that alters modern out-of-order processors by introducing a transactionally read bit in the L1 cache and by holding speculative writes in the store buffer. The system issues exclusive requests when the processor retires a tentative store, discarding and restarting the computation of those processors that owned the line. In case of failure, the critical section retries its execution using locking semantics. Once all the exclusive permissions are granted, processors atomically drain the store buffer into the data cache while keeping denying incoming requests. Thread Lock Release (TLR [96]) extends SLR to construct a fair

HTM System	Eager HTM Group	Access Summary	VM Strategy	Bounded Support	Finite Tx?	Overflow Support
Original HTM [49]	Bounded HTM	R/W L1 bits	Late	Separate Tx L1 cache	Yes	None
RockHTM [33]	Bounded HTM	Read L1 bits	Late	Store Buffer	No	STM
HASTM [105]	Hardware Accelerated	Read L1 bits	STM	Tx L1 cache	No	STM
Damron's HyTM [31]	Hybrid TM	R/W L1 bits	Late	Tx L1 cache	No	STM
VTM [97]	Unbounded HTM	R/W L1 bits SW filters	Late	Tx L1 cache, Tx registers	No	SW buffers
LogTM-SE [130]	Unbounded Log-based HTM	Signatures	Eager	None	No	SW log

Table 2.1: Classification of eager HTM systems

timestamp-based algorithm that enforces younger transactions to restart or wait. This starvation-free algorithm guarantees that at least one of the conflicting transactions keeps doing progress.

Since SLE, some store-buffered HTM systems have been proposed. Such late version management scheme is promising given it requires minor modifications in the hardware, especially in memory structures. In fact, the Rock processor [33] prototyped by Sun Microsystems exploits a flash copy mechanism to checkpoint the architectural register state and limits the version management support to the size of the store buffer. It also exposes the hardware resources to the software layer, which is notified each time a transaction fails. (This includes buffer overflows, data races between transactions or processor exceptions and interruptions.) In those situations, the system can behave like a HyTM, as it implements a friendly interface for STM [79].

2.3.2 Hardware-accelerated TM Systems

Saha *et al.* proposed the acceleration of *software* transactions through specific hardware mechanisms [105]. In their HASTM implementation, they exposed four L1 cache bits to the architecture, allowing the software to monitor, test and clear those bits. Thus, the STM layer was able to track transactional evictions and invalidations, reducing the software cost of managing extra metadata such as the whole read set.

RTM [113] implements an object-oriented RSTM that modifies the cache coherence protocol with 5 additional states to hide transactional actions in the L1 cache. Although this novel approach increments the complexity of the system, it also minimizes the overheads associated with the access summary and data versioning mechanisms. RTM's software controlled coherence permits the system to monitor any communication among processors. Hence, when the software discovers an inconsistent access, it has to notify conflicts to individual processors. This is made through the Alert-On-Update mechanism, which extends the ISA with additional instructions to expose convenient information (*e.g.*, when to abort) to independent processors.

SigTM [17] augments a TL2 STM system by substituting slow metadata structures with hardware (read and write) signatures. Note that this keeps caches clean—read and write sets are not maintained in metadata software structures or in the L1 cache. In SigTM, each transactional write has to execute an additional instruction to introduce its address in the signature. Similarly, a transactional read has to test remote write signatures when it asks for the exclusiveness of a line. As write-write concurrency is allowed, transactions must validate their write set at commit time. This is done by issuing special exclusive requests, which may not be granted if an address belongs to another in-flight transaction. If these special instructions fail, the software takes command and resolves the conflict.

Dalessandro *et al.* showed how a NOrec STM system can coexist—and, of course, accelerate their execution—with Rock-like HTM support [30]. For example, hardware and software transactions use a nested hardware transaction (two hardware-accelerated writes) to acquire a secondary lock needed to validate the committing values. This fact prevents transactions that fit in the hardware to interfere with software transactions.

2.3.3 Hybrid TM Systems

Hybrid Transactional Memories (HyTM) handle large transactions using software mechanisms such as STM [48] systems, whereas common-case, smaller transactions use best-effort hardware. Kumar *et al.* extended core resources with a transactional state table and a highly-associative transactional buffer in order to simultaneously combine hardware and software transactions [63]. The first records the execution mode of each in-core hardware context and its associated transaction, while the latter stores, for each cache line, both the old and new values and a vector of R/W bit—one per each hardware context executed in the core. If virtualization is needed the system aborts the current transaction and restarts it in software (DSTM) mode.

Damron *et al.* cut apart the software strategy from the hardware support beneath the HyTM system to embrace a wider range of applications, but software transactions may slow down hardware transactions due to additional lookups [31]. PhTM [65] increments the flexibility of hybrid environments by enlarging the modes of execution and by preventing hardware-only transactions to overlap their execution with software-only transactions.

Hybrid UFO [8] expands the system with fine-grained hardware memory protection to achieve strongly-atomicity between STM/HTM transactions and outsider code. Moreover, transactions that fit in special-purpose hardware can run safely with concurrent software transactions, although multiple readers of a memory block are not allowed. Exposing potential conflicts among transactions to the software permits the system to apply fair contention management policies, such as age-based approaches—these policies prioritize, in the common case, the STM transaction. The study of distinct software conflict management policies also pointed out that it is important not to fail over the STM when contention is high.

MetaTM [98] modifies x86 architecture to provide HTM assistance, allowing the system to execute a “transactified” release of Linux, called TxLinux. MetaTM upholds multiple methods for resolving conflicts between transactional accesses. Polka, a policy that aborts the transaction that has done less work—*i.e.*, has executed less instructions—and performs an exponential backoff before restarting it, is the one that achieves the most average performance, although it is not the best in all the cases. In [54], MetaTM simplifies the transactional hardware, at the cost of limiting concurrency. It proposes *transactional ordering*, a mechanism that relies on a runtime system that assigns kernel-level locks to those transactions that overflow physical resources and commits them in serial order.

Riegel *et al.* [101] expanded the design space of HyTM systems by implementing time-based algorithms using AMD’s Advanced Synchronization Facility (ASF [27]), an x86 ISA extension that aims to provide rich semantics for easing the synchronization of threads.

2.3.4 Unbounded HTM Systems

Bounded HTM systems do not ensure by themselves the progress of those transactions that exceed buffering resources, such as local caches. However, high-performance HTM systems must support in hardware transactions of arbitrary size, even in the case of overflow. While HaTM and HyTM systems offer clean solutions when dealing with those situations, software dependency induces substantial performance overhead. To this end, Ananian *et al.* proposed Unbounded TM (UTM [5]), a hardware-assisted system where *all* information regarding a transaction is held in a unique memory-resident data structure.

Since UTM introduces several changes in processor's state, UTM authors presented a simplified (almost unbounded) HTM implementation, called Large TM (LTM [66]), that allows a safe execution of transactions as large as physical memory. LTM reserves part of its non-cached DRAM memory space—organized as a hash table—to buffer updated data spilled from the L1 cache. Note that LTM employs *late* data versioning, keeping the pre-transactional state in the shared levels of the memory hierarchy.

Virtualized Transactional Memory (VTM [97]) maintains vital information of data that exceeds hardware resources—*e.g.*, speculative values or read and write sets—in a table placed in application's *virtual* memory. Similarly, Page-based Transactional Memory (PTM [24]) expands a conventional bounded HTM system with shadow pages that hold transactionally modified values. Those deferred update VM schemes, however, experiment large delays when overflowing data is made visible at the end of a transaction. To overcome this issue, lots of HTM implementations offer *early* data versioning to manage efficiently the speculative (new) transactional state. In those approaches, hardware structures that hold new values are replaced with low-cost mechanisms (commonly software-resident logs) to keep the transactional state in-place and the pre-transactional (old) state in private (per-thread) memory.

Moore *et al.* were the first to simplify the UTM's version management mechanism by using software-resident logs to maintain the old values of transactionally written lines. In their LogTM implementation [84], the software log is logically organized as a stack, placing the old values of recent modified data above the values of previously accessed data. In case of abort, a software routine traversed the log to undo the modifications introduced by the offending transaction. While LogTM only supports flat nesting [43], optimized reinterpretations of the original system [85] enable composing transactions by encapsulating the nesting depth and a pointer to its parent in the header of the software-resident log.

As a result of its simplicity, the majority of contemporary HTM proposals [70, 122, 130] establish *early* software logging as their data versioning strategy. These implementations are ordinarily known as log-based HTM systems. This method relocates transactionally modified data across whatever level of the memory hierarchy, which means that just a single speculative version of a memory block can be spilled in the shared memory space. Thus, log-based HTM systems impose eager conflict management, building up what is commonly known as **truly eager** (*early* VM, *eager* CM policies, *EE* for short) HTM systems. In Section 3.2.2 we describe LogTM-SE [130], an evolution of LogTM that replaces R/W bits with signatures to summarize those *physical* addresses accessed within a transaction. We also explain in detail the logging and abort process on conventional log-based HTM systems. Section 4.8 describes in more detail related work on unbounded and log-based HTM systems.

There are several *metadata* HTM implementations that combine software-resident logs with per-bock memory extensions for access summary purposes. OneTM [10] introduces a permission-only cache to maintain consistency of evicted cache lines. The system just allows one overflowing transaction at a time, which can be executed in parallel with multiple small transactions. Overflowed lines must keep a transaction identifier, which determines which thread owns—and thus can access—the evicted data.

Similarly, TokenTM [12] eliminates the false positives of signatures by adapting the concept of token coherence to detect conflicts among any kind of transactions. In TokenTM, each memory block carries T tokens, which are propagated through the memory hierarchy using metabits. Threads must acquire one (all) token(s) of each memory block read (written) within a transaction. This fact does not generate any ordering issue in TSO-like implementations given that TM applications are *safe supervised* [13]. If a token cannot be acquired, a conflict is detected and the transaction must abort. Tokens are restored at commit or abort time, using metadata *fusion* and *fission* techniques to accelerate the process. LiteTM [58] reduces the number of out-of-band bits required to represent the token state by exploiting locality properties and using software to infer the lost information.

2.4 Lazy HTM Systems

Some HTM implementations defer the resolution of conflicts until commit time. As these approaches permit inconsistencies among transactions, multiple versions of the same line have to be maintained in hardware. Hence, *late* version management is a must when designing those HTM systems. That is the reason why these systems are commonly known as **truly lazy** (*late* VM, *lazy* CM policies, *LL* for short) HTM systems. Resolving conflicts after they are produced enables speculation between contended transactions, which favors overall concurrency and eliminates some read-write conflicts. However, arbitration and atomic data movement is required when a transaction finalizes. In Section 5.5 we survey in more detail how lazy commit protocols operate.

Stone *et al.* proposed Oklahoma Update [116] to replace short critical sections with multi-word atomic updates, the values of which were stored in multiple (up to 8) reserved registers. These registers buffered updates until commit time, where the processor requested exclusive write permissions of the accessed addresses. Once all the permissions were acquired, the processor sent write requests to memory, blocking incoming request to prevent interruptions. This process is commonly known as *two-phase* commit—acquire permissions (*first* phase), then update memory (*second* phase). To avoid deadlocks, permissions were acquired in ascendant address order.

Transactional Coherence and Consistency (TCC [45]) presented a new shared-memory model based on transactions. In TCC, *every* operation is performed inside an atomic block declared statically by the programmer or the compiler; making transactions the basic unit of work from the system point of view [44]. Thus, memory accesses performed inside a transaction can be freely re-ordered, given that they will appear atomic from outsider threads. This concept is generally known as *transactional* consistency. Table 2.2 summarizes some lazy HTM implementations with TCC background.

In contrast to conventional eager HTM systems, lazy HTM implementations require several changes in the memory hierarchy and in the coherence protocol. A TCC-like system must have at least two levels in the memory hierarchy, one private that keeps the *speculative* state—in the original TCC, R/W bits are added to track transactional lines—and one shared that holds the

HTM System	Lazy HTM Group	Access Summary	VM Strategy	Bounded Support	Finite Tx?	Overflow Support
TCC [45]	TCC-based HTM	R/W L1 bits	Late	Tx L1 cache, Central Arbiter	Yes	Grab Token
XTM [25]	TCC-based HTM	R/W L1 bits, Snapshots	Late	Tx L1 cache, Central Agent	No	Trap Exception
Bulk [20]	Bulk Consistency	Signatures	Late	Tx L1 cache, Central Arbiter	No	DRAM Space
FlexTM [112]	Hybrid TM	Signatures	Late	Tx L1 cache	No	STM

Table 2.2: Classification of lazy HTM systems

committed (non-speculative) state. In-flight transactions read data from the shared (or global) state, and hold new values locally until the transaction ends.

The global state is *atomically* updated at commit time following Oklahoma's two-phase protocol, and only one transaction can commit at a time. Thus, it is necessary to arbitrate between in-flight transactions, allowing a single committer in the system. TCC opts to integrate in the hardware a centralized agent that fairly distributes a global token among transactions willing to commit. If a transaction fails to acquire the token, it must wait until the token is released [82]. Once a transaction acquires a token, it broadcasts its write set to all the processors, which inspect their access summary to find inconsistencies. If so, non-committers abort their running transaction. After that, the committer updates the shared memory. As the commit is an atomic process, the system must block accesses to the modified lines until the commit ends—*i.e.*, after the committer releases the token.

If a transaction does not fit in the L1 cache, the overflowing processor must acquire the commit token and hold it until commit time, so the transaction can safely update shared memory with the speculative state. This mechanism may starve younger committers, which cannot acquire the token and thus commit, serializing parallel execution. To avoid this problem, Extended Transactional Memory (XTM [25]) complements TCC with a page-based virtualization strategy that buffers updates in private pages and uses snapshots for conflict detection. TCC can also be enlarged with nesting, I/O and OS support [81], as well as word-granularity conflict detection. Enhanced TCC-based systems can rely on programmers' ability to leverage the commit packet [107].

Ceze *et al.* compact memory accesses executed within atomic blocks in finite signatures for different purposes, ranging from TM or TLS [20] to high-performance sequential consistency [21]. Specific hardware is necessary in order to efficiently operate with signatures. Like in TCC, collisions between transactions (chunks of instructions) are resolved at commit time by broadcasting the write signature and performing local Bulk disambiguation. In the case of violation, the transaction (chunk) is aborted (squashed) and immediately re-executed.

Several improvements have been proposed to enhance the scalability of lazy HTM systems [92, 93, 112, 124]. These techniques are broadly discussed in Chapter 5. An interesting alternative to rethink a lazy system is FlexTM, which uses a software layer to define the resolution of conflicts while the hardware is in charge of maintaining the speculative state. We review FlexTM in Section 6.6 together with related work on contention-aware HTM systems.

2.5 Reutilizing Transactional Mechanisms

Few proposals extended HTM support for non-transactional purposes, such as enforcing sequential consistency in weaker memory models, checking determinism, detecting races between atomic sections or maximizing the performance of sequential and parallel execution.

Invisifence [11] pins down non-ordered data in private caches and supervises memory messages to support sequential consistency among parallel chunks of instructions. A chunk ends when it receives cache permission for all the memory instructions contained in it. Similar to eager HTM systems, Invisifence clears speculative bits from caches and flushes processor's pipeline in case of collision between in-flight chunks, avoiding weak-ordered executions.

Porter *et al.* adapted hardware-assisted (either eager or lazy) TM support to accelerate sequential execution in a Speculative Multithreaded (SpMT) environment [91]. This approach achieves higher concurrency between speculatively spawned threads, given that they can optimistically execute in parallel in the (common) case that memory dependences do not appear. Like this, SpMT implementations can be more aggressive than conventional designs.

TM support can also be used for testing parallel code. StealthTest [15] exposes software transactions as the principal operation to undo the inherent modifications introduced by on-line testing of parallel code. RaceTM [42] detects data races between ongoing threads using TM-like coherency to report *bugs*—*i.e.*, conflicts between implicit transactions—on non-protected parallel code. LifeTx [60] enforces deterministic thread interleaving by encapsulating parallel code inside bounded and ordered hardware transactions.

Chapter 3

Experimental Methodology

This chapter overviews the simulation infrastructure utilized in the next chapters to evaluate the contributions of the thesis. After that, it details the base system configuration used through the study and presents the state-of-the-art, reference HTM systems against which the proposed techniques are compared. The chapter ends describing and characterizing the transactional benchmarks used for the evaluation and presenting the performance metrics employed to analyze the behavior of the proposed HTM designs.

3.1 Simulation Infrastructure

A complete system has been simulated using the Simics [76] infrastructure from Virtutech and the GEMS [80] toolset from Wisconsin's Multifacet group. Simics is a commercial product that provides full-system functional simulation of a multiprocessor system executing a SPARC Instruction Set Architecture (ISA) [55]. This environment enables the evaluation of TM workloads running on top of a Solaris 9 Operating System (OS). GEMS (version 2.1) is a timing module that has been used to model the memory hierarchy, the HTM base systems, the coherence protocol and the network traffic. GEMS is essentially written in C++, but it uses SLICC—a specific domain language—to define coherence transitions and intermediate (non-solid) cache states.

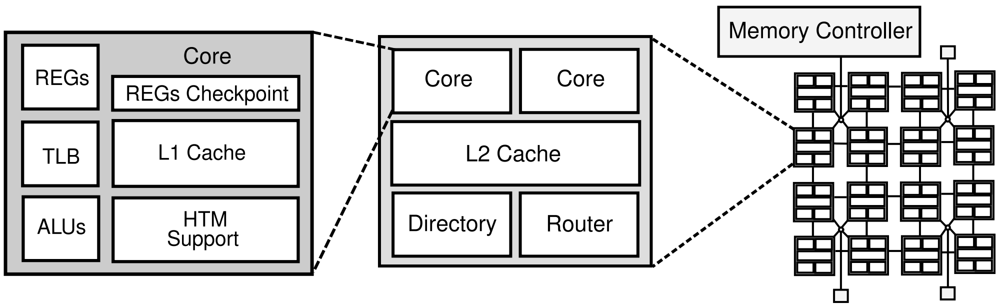


Figure 3.1: Base system configuration

3.1.1 Modeling Hardware Support

Simics processing cores retire an instruction per cycle for non-memory operations. However, in our infrastructure, those cores communicate with GEMS to model the latency of memory operations. Each time a core retires a memory operation, it blocks its execution and it delegates the management of memory to GEMS, which simulates the timing of the operation (coherency included) and updates the memory state before returning the government to Simics.

GEMS treats non-included ISA instructions as "magic" instructions—*i.e.*, a SPARC no-operation (NOP) when they are executed in a real machine. When a processing core issues a magic instruction (*e.g.*, `Tx_Begin` or `Tx_End` in a TM habitat), it gives control to the GEMS timing model, which prepares the processor state for transactional purposes. Transactional operations are handled as regular memory operations—the only overheads introduced in the timing model are those produced by special coherency events.

Notice that GEMS is possibly the most used infrastructure in the literature to model HTM systems due to its flexible capabilities. Several HTM systems [84, 112, 122, 130] have been built on top of this simulation environment. Its adaptability allows the system to efficiently pass information to the software using hidden processor registers and trapping precise exceptions appropriately.

3.2 System Configuration

For our evaluation, we assume a Chip Multiprocessor (CMP) with 32 cores and two levels of caches, where the first level (L1) is private and the second level (L2) is shared among all the

Core	32 UltraSPARC III cores, 1.2 GHz in-order, IPC = 1, single issue, single-threaded
L1 cache	32 KB, 4-way, 64 bytes per line, inclusive, write-back, 2-cycle latency
L2 cache	16 MB, 8-way, banked NUCA, write-back, 15-cycle latency
Memory	4 GB, DRAM, banked, 150-cycle latency
Memory controllers	4 Memory controllers, distributed in the CMP, 25-cycle latency
L2 directory	Bit vector of sharers/owners, distributed, L2 inclusive, 6-cycle latency
Interconnect	16-node mesh, 64-byte links, 2-cycle wire latency, 1 cycle route latency
Base HTM Support	2 Kb parallel Chuckoo-Bloom filters Register checkpoints per core Software logging support

Table 3.1: Base system parameters

cores, as shown in Figure 3.1. Coherency is implemented using a blocking, distributed directory placed in the L2 cache. The system has a 16-node mesh interconnect that uses 64-byte links with adaptive routing. Each node has two cores, a piece of a shared L2 cache and part of the directory. Further information about simulation parameters is described in the next subsection.

3.2.1 Base CMP Parameters

The CMP models 32 simple, single-threaded, in-order `UltraSPARC III` cores with fixed IPC 1 for non-memory operations. Memory operations, instead, take variable latency.

Each core has two private 32KB cache, one for instructions and one for data. The L1 data cache is write-back and coherent. Besides the standard logic, cores are extended with additional hardware support for TM. For example, cores use local shadow copies of physical registers, which are updated each time a transaction starts and are used to recover the processor state when an abort occurs. We defer the description of specific in-core hardware support for each

HTM to the following section (for reference HTM systems) and chapters (for proposed HTM systems).

Our base system implements a shared L2 cache distributed among the CMP nodes [50, 67], where each node has 1 MB of the L2 cache. This is a Non-Uniform Cache Access (NUCA) system that contains a total of 16 MB. The system has four memory controllers to access 4 GB of main memory. Coherency among L1 caches is implemented on top of a split directory placed at the L2 cache, which keeps, for each line, a list of sharers and owners (if more than one). Table 3.1 contains additional parameters of the base CMP system.

3.2.2 Reference HTM systems

We take two state-of-the-art HTM systems as baseline architectures to evaluate the effectiveness of *truly* eager (LogTM-SE) and *truly* lazy (TCC) schemes. Following is a description of the reference HTM systems.

LogTM-SE [130]: Our eager start point is LogTM-SE, a log-based (EE-like) HTM system developed at the University of Wisconsin-Madison and distributed with GEMS 2.1. This implementation stores directly transactional values in Simics memory and delegates the detection of conflicts to a *filtered* MESI coherence protocol, which forwards coherence messages to the owner (or sharers) of a cache line. Memory locations accessed within transactions are kept in per-core read and write signatures, which are consulted each time the core receives a remote memory request.

For our evaluation, we choose to use 2Kbit parallel Cuckoo-Bloom filters [130], given that previous studies advocate that this configuration commonly obtains the best performance-per-area results [106]. Like all log-based HTM systems, LogTM-SE initializes the software-managed log when a transaction begins—*i.e.*, when a processor retires a `Tx_Begin` instruction. In such situations, the processor jumps to a firmware routine that initializes the log.

In LogTM-SE, each time that a memory operation misses the L1 cache the system must perform conflict detection by issuing a memory request to the memory subsystem. If the memory request does not report a conflict, the reader (writer) core adds the memory address in its Read (Write) Signature. In addition to this, when a transactional store is retired (*i.e.*, no conflict appears) the core must maintain the old value of the line in a private software log before writing the new value to memory. The software log contains, for each transactional written line, the *logical* address of the line and its pre-transactional value [88].

Our LogTM-SE implementations follows three steps to ensure that the new value is in place and the old value in the software log: (1) the system brings the cache line to the processor if it is not already there, checking for conflicts through forwarded coherency requests, (2) if there is no conflict, the old data is stored in the first available entry of the stack together with its logical address and (3) the new data is stored in the L1 cache and the log pointer is incremented. Note that the logging process can be treated as a conventional non-transactional store, so conflict checking is not required—the software log is private and thus not accessed by other processors.

When a core receives a conflict notification, it re-issues the memory operation again, hoping the conflict to disappear soon. However, LogTM-SE may abort a transaction when a cycle among conflicting requests is detected. When a core has to abort, it has to undo all the changes performed by its transaction. This is done by triggering an exception that jumps to a recovery handler. This handler invokes a software routine that walks the log in reverse order and, for each undo entry, stores the old data at the address associated with that entry. When the first entry of the log is restored—the routine arrives at the head of the log—the handler informs the hardware that it can clear the access summary, recovers the register checkpoint and sets the PC to the value stored in the header log, which corresponds to the start point of the transaction.

As transactions are durable, once they commit their changes are preserved forever—*i.e.*, the stacked data is discarded by updating the log pointer. Our implementation extends the base conflict resolution policy of LogTM-SE with other policies described in the literature [14, 98].

TCC [45] with distributed commits [92] (TCC-Dist): Modeling TCC-based (LL-like) systems may turn into a complex task because the majority of proposals use *ad hoc* HTM implementations. Nonetheless, we integrate *truly* lazy execution in a conventional CMP environment by adjusting the coherence protocol to allow multiple versions of the same cache line. The complete design of this system is described in Chapter 5.

Lazy HTM systems do not update memory until a transaction commits, so our lazy implementation modifies the simulation interface to hide transactional stores from Simics, keeping old values untouched in global memory and new values only locally visible. Hence, the simulator buffers transactional writes and bypasses (updates) these values each time a younger reader (writer) accesses the memory block within that transaction. Despite TCC does not support unbounded transactions, we idealized late VM hardware to permit the execution of transactions of any size. This implementation allows us to emulate the optimal behavior experimented in FlexTM [112] or EazyHTM [124], which mitigate the impact of resource overflow by using specially designed hardware.

Like FlexTM, our lazy approach relies on signatures to track the read and write sets. To provide a fair comparison, we used the same signature parameters than in LogTM-SE. Contrary to FlexTM, our base lazy HTM system does not require software arbitration to guarantee transactional consistency. Instead, we borrow the distributed technique presented in [92] to enable parallel (and reliable) commits. To implement this technique, each core keeps a bit vector containing all the directory banks accessed during the transaction. Before making the state visible (*i.e.*, moving the speculative state to non-transactional), the committing core must acquire all the directories that are present in the read and write sets. When a core fails in its attempt (this happens when another transaction is committing a transaction that has accessed the same directory), it must re-issue the *Acquire* message. Directory steals are allowed in order to prevent directory deadlocks. After acquiring all the directories, the core sends abort messages to conflicting cores, updates Simics global memory and releases the directories. More details about the base commit process (and its optimizations) are described in Chapter 5.

3.3 Transactional Workloads

Evaluating HTM systems may become a labyrinthine task given the lack of TM software developed until the date. The majority of HTM systems appeared in the literature do not contemplate complex TM workloads (in part because they did not exist by the time those systems were published), and most TM behavioral studies are based on simple programs that read/update shared variables (*e.g.*, global counters) or small data structures. Instead, we characterize both reference and proposed HTM systems with a vast range of TM applications, which allows us to better prove the (dis)advantages of each approach.

All the TM benchmarks are multithreaded applications written in C. These applications first prepare the input data for computation and then divide the work in independent threads,

Suite	Benchmark	Input parameters	Execution
μbench	Btree-fix	10/90% inserts/lookups, fixed Tx size	16K iterations
	Btree-var	50/50% inserts/lookups, variable Tx size	2K iterations
	List-long	5K dummy work, 2K useful work, 16 lists	4K iterations
	List-short	2K dummy work, 1 useful work, 1 list	16K iterations
	Hash-read	10/80/10 inserts/lookups/deletes, 4K buckets	4K iterations
	Hash-write	25/50/25 inserts/lookups/deletes, 1K buckets	4K iterations
SPLASH-2	Barnes	512 bodies	Whole parallel phase
	Raytrace	Teapot	Whole parallel phase
STAMP	Bayes	32 vars, 1024 records	Whole parallel phase
	Genome	32K segments, 512 genes, 32 lengths	Whole parallel phase
	Kmeans-low	40/40 clusters, 16K points	Whole parallel phase
	Kmeans-high	15/15 clusters, 16K points	Whole parallel phase
	Intruder	4K traffic, 10 attack, 4 packs	Whole parallel phase
	Labyrinth	32*3*3 maze, 2048 routes	Whole parallel phase
	Ssca2	2^{13} nodes, 3 edges, 3 length	Whole parallel phase
	Vacation-low	1M clients, 90% queries, 4 items	16K tasks
	Vacation-high	64K clients, 60% queries, 16 items	4K tasks
	Yada	20 angle, 633.2 mesh	Whole parallel phase

Table 3.2: Input parameters of TM applications

which are bound to a unique processing core using `pthreads`. Threads are executed in parallel, accessing shared data within transactions. Before starting the parallel phase, a Simics "magic" instruction is introduced in order to warm up transactional structures. A non-blocking memory pool library is used to remove implicit locking from OS operations (`malloc` or `free`) performed inside transactions. Global barriers are placed at the end of the computation to synchronize threads. After that, the master thread performs sanity checks.

3.3.1 Transactional Benchmark Suites

TM is an emerging programming paradigm, and thus there is not a quorum regarding which is the best-suited methodology to quantify HTM systems. This partly happens because it lacks a standard TM benchmarks suite, so most proposals are evaluated with self-developed programs. We believe that this is a not-fair strategy, given that it may be applied at convenience, leading to incorrect conclusions. In order to run away from those tricky methods, this dissertation evaluates its contributions with a wide spectrum of TM applications, which vary in terms of transaction's length and contention. The TM benchmark suites utilized are described below.

Microbenchmarks [80] are slightly modified benchmarks which plain version is provided by GEMS. These benchmarks interact with distinct data structures and present variable contention, depending on how data is distributed on those structures. We rewrote those benchmarks (*Btree*, *List* and *Hash*) to regulate (by input parameters) overall transactional size and contention. We utilize these benchmarks to stress the HTM systems, which permits us to distinguish clearly (and rapidly) the strengths and weaknesses of each implementation.

Splash-2 [128] presents a set of benchmarks for multiprocessors, where lock-protected regions are replaced with transactional blocks. As Splash-2 benchmarks have been tuned over the years to minimize synchronization, they spend most of the time in small, fine-grained transactions. Although this behavior is not TM representative, we included in our evaluation two Splash-2 benchmarks (*Barnes* and *Raytrace*) because (i) they enable a fair comparison with past HTM work [130], which used this benchmark suite for their evaluation, and (ii) they concentrate shared data in few code lines, what makes critical the conflict management policy implemented in the system.

STAMP [16] is the first (and, until the date, the unique) pure transactional benchmark suite. STAMP workloads try to recreate how an average programmer would implement an application using a TM programming model. In these workloads (*Bayes*, *Genome*, *Intruder*, *Kmeans*, *Labyrinth*, *Ssca2*, *Vacation* and *Yada*), programmers conservatively protect accesses to shared data structures in large (even huge) transactions, which eases programmability but increases the conflicting rate—and thus scalability. Hence, these workloads spend most of the time running transactions, what raises interesting performance concerns when they are executed in non-optimized HTM systems. For the evaluation, most STAMP workloads use the input parameters suggested in the 0.9.10 distribution. However, some input parameters are modified (using values suggested for non-simulated executions, *i.e.*, STM systems) to preserve the scalability of TM applications.

Category	Benchmark	Avg. Read Set	Avg. Write Set	Max. Read Set	Max. Write Set
Fine grain	Barnes	6.27	4.63	41	35
	Kmeans-low	8.00	3.50	10	4
	Kmeans-high	7.25	2.75	9	3
	List-short	1.25	1.50	2	2
	Raytrace	5.32	1.98	458	3
	Ssca2	3.00	2.00	3	2
Variable grain	Bayes	79.29	37.73	806	455
	Btree-var	155.25	85.24	291	262
	Genome	31.74	10.47	155	45
	Hash-read	128.73	121.19	282	270
	Intruder	8.71	2.97	43	21
	List-long	30.81	30.67	261	259
	Yada	32.77	14.85	326	158
Coarse grain	Btree-fix	36.44	13.52	51	23
	Hash-write	158.19	150.25	401	389
	Labyrinth	112.02	102.16	316	222
	Vacation-low	97.47	20.77	184	31
	Vacation-high	101.72	20.20	234	37

Table 3.3: TM applications grouped by the size of their transactions

All transactional benchmark suites are compiled with `gcc 3.6` using the -O2 optimization flag. They are also executed until their completion taking as an input the parameters described in Table 3.2.

3.3.2 Transactional Workload Characterization

Prior studies assumed that transactions are commonly small and do not conflict [26]. However, novel TM workloads include large transactions that access shared data structures, what may often produce collisions. The performance achieved by HTM systems is highly application dependant; therefore it is interesting to categorize TM workloads according to their characteristics. Grouping applications permits a comprehensible understanding of how transactional

Category	Benchmark	Transactional Time	Committed Transactions	Tagged Transactions
Barely Tx	Barnes	2.11%	2187	3
	Kmeans-low	3.65%	21846	3
	Kmeans-high	8.77%	21846	3
	Intruder	29.30%	22501	3
	List-short	2.38%	32768	2
	Raytrace	0.14%	47751	5
	Ssca2	14.45%	47257	3
Mostly Tx	Bayes	81.79%	490	15
	Btree-var	98.86%	2048	2
	Btree-fix	91.79%	16384	2
	Genome	97.57%	19483	5
	Hash-read	98.47%	4096	4
	Hash-write	98.78%	4096	4
	Labyrinth	99.48%	4098	3
	List-long	62.74%	8192	2
	Vacation-low	91.15%	16384	3
	Vacation-high	86.34%	4096	3
	Yada	99.92%	2788	6

Table 3.4: TM applications grouped by the size of their transactional time

mechanisms operate under certain scenarios and allows us to determine the major performance bottlenecks associated with these situations.

Table 3.3 provides important information about the size of the transactions that belong to the applications utilized in this dissertation. For each benchmark, Table 3.3 shows a column with its average transactional read set size (Avg. Read Set), the average transactional write set size (Avg. Write Set), the maximum read set size (Max. Read Set) and the maximum write set size (Max. Write Set) of a single-threaded LogTM-SE execution (cache line granularity). The last column classifies applications in three different categories according to the data gathered in the previous columns.

We refer as *fine-grained* applications those that contain transactions that read and/or modify few lines. *Variable-grained* applications are those that combine small and huge transactions to build software using rational-size atomic blocks. These applications are easy to identify because average and maximum read and write sets considerably differ. In contrast, *coarse-grained* applications are dominated by large transactions that cope most of application's execution time. Variable- and coarse-grained applications recreate the (expected) behavior of future parallel software, where non-expert programmers enclose shared data utilizing conservative principles.

Table 3.4 complements the previous table with additional information of single threaded LogTM-SE executions. It shows the percentage of time spent inside a transaction (column Tx Time), the number of committed transactional instances during the application (column Committed Tx) and the number of *different* tagged transactions executed in the application (column Tagged Tx). The last column of Table 3.4 classifies the benchmarks in function of its transactional weight.

We define as *barely transactional* those applications that spent most of the time in parallel, independent computation or in barriers waiting other threads to finish their computation. These applications may experiment high contention given that they tend to concentrate shared memory accesses in tiny transactions. Instead, *mostly transactional* applications cover almost all parallel computation using transactions.

Prior tables provision a summary of static transactional information. Unfortunately, this data is not sufficient to extract conclusive assumptions of application's scalability capabilities. For example, someone could claim that fine-grained, highly-tuned applications should perform better than coarse-grained applications. This is commonly true in STM systems, where transactional overheads downgrade considerably overall performance [18]. However, in HTM systems, fine-grained applications usually concentrate data collisions in specific points of the program, which may generate high contention and thus worse performance.

All the above has lead us to measure the contention level of the applications. Table 3.5 summarizes contention information regarding 8-threaded LogTM-SE executions. The second column of the table (Conflict Rate) shows the number of collisions per committed transaction, while the third column (Abort Rate) shows the number of aborts per committed transaction. The forth column (Contention Overhead) shows the percentage of time spent managing contention—

Category	Benchmark	Conflict Rate	Abort Rate	Contention Overhead
Low Contention	Hash-read	0.48	0.07	28.32%
	Kmeans-low	0.01	0.001	0.02%
	Kmeans-high	0.03	0.002	0.15%
	Raytrace	0.08	0.05	0.75%
	Ssca2	0.005	0.001	0.78%
	Vacation-low	0.07	0.001	2.70%
Medium Contention	Barnes	0.32	0.24	6.99%
	Btree-fix	0.04	0.03	12.17%
	Genome	0.13	0.08	32.07%
	List-short	0.25	0.18	5.84%
	Vacation-high	0.12	0.02	10.47%
High Contention	Bayes	2.93	2.26	78.60%
	Btree-var	0.23	0.19	36.77%
	Hash-write	1.50	0.39	56.81%
	Intruder	4.37	2.95	71.06%
	Labyrinth	0.85	0.76	63.50%
	List-long	0.62	0.40	37.35%
	Yada	2.55	1.79	77.74%

Table 3.5: TM applications grouped by the size of their transactional contention

i.e., time spent executing discarded work, re-issuing a memory request, recovering the pre-transactional state and spreading transactional computation.

The last column of the table classifies the applications according to the contention obtained using our baseline HTM system. *Low-contention* applications experiment few conflicts, so they are expected to scale well even with more threads. *Medium-contention* applications show variable contention through different phases of the execution, what should limit global performance as reported by Amdahl's Law [36, 52]. Finally, *high-contention* applications present a great challenge for HTM systems, given that they are expected to perform poorly due to the huge overhead produced by conflicting memory accesses.

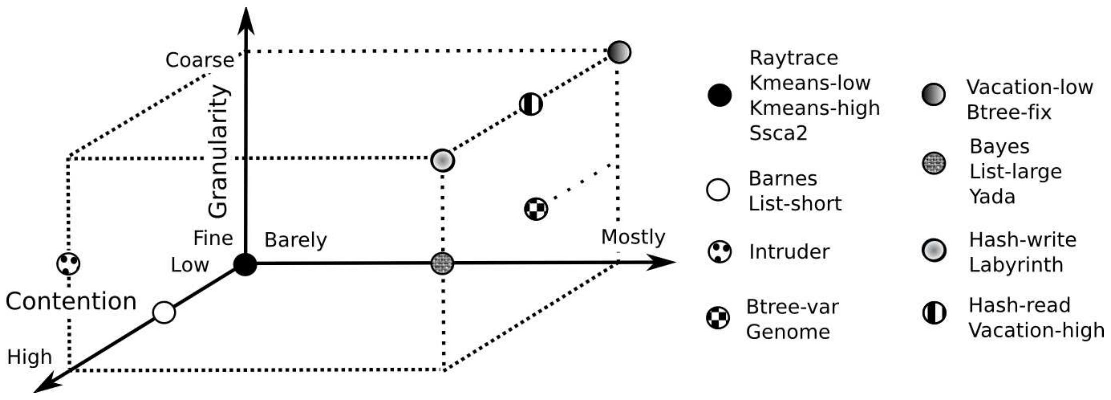


Figure 3.2: Clustering TM workloads according their characteristics

3.3.3 Discussion about Transactional Workload Behavior

Figure 3.2 clusters TM applications using the categories introduced in Tables 3.3- 3.5. As it can be seen in the figure, most transactions rather differ in the way they use transactions, implementing transactions of distinct size, time and contention. Performance analysis carried out in this dissertation are grouped in accordance with one (or more) of the above categories. This classification allows the reader to better understand the limitation factors of each HTM system when it executes a specific type of application.

Thus, we want to note that a heterogeneous set of applications has been used to evaluate the proposals presented in this dissertation. Our classification indicates that most of them exhibit different properties, and it seems reasonable to think that they will show a divergent behavior when they execute over the same underlying hardware. For example, *Labyrinth* and *Vacation-low* are both coarse-grained applications, but possibly only *Vacation-low* will scale in LogTM-SE due to its low contention.

Similarly, these applications may alter their performance when they are executed on distinct HTM systems. As discussed in the previous chapter, lazy HTM systems commonly present poor scalability when they execute fine-grained applications [16], like *Ssca2*, because of the overhead introduced on commits [45]. However, they also favor concurrency in high-contention applications [111]. Hence, some high-contention applications with small transactions, such as *List-short* or *Genome*, may obtain better performance when they are run on a lazy HTM system.

3.4 Performance Metrics and Methods

Evaluating HTM systems is an extremely sensitive labor, as common performance metrics can not be applied. For example, an HTM system with high IPC may present ridiculous performance if retired transactional instructions are later discarded when transactions abort. Similarly, specific contention metrics, such as *Abort Rate*, may produce tricky behavioral pathologies. For instance, a high-contention TM application may show a low *Abort Rate* but poor performance if processors are disabled after abort recovery.

For our quantitative performance evaluation, we focus on the parallel phase of the program, skipping the initialization and the end phase. Hence, we run the TM applications until their end, taking the overall execution time as the basic metric for the performance analysis. Notice that threads are synchronized with barriers at the end of the program, ergo the execution completes when the last operative thread finishes its computation. In our performance analysis, execution time is normalized according to the reference HTM systems to ease the comparison with other HTM proposals.

Although normalized execution time is necessary to show *how much* performance each HTM system obtains, it is even more important to understand *why* each HTM system achieves that performance. Thus, is crucial to distribute the execution time to see which computation is useful and which is discarded. Our analysis does such distribution in a way that permits a fast localization of HTM main limitations.

Besides execution time, this dissertation uses secondary metrics to provide further information regarding transactional behavior. These metrics complement the analysis by pointing out specific factors that may impact overall performance. The abort rate, commit contention or number of network messages are few examples of secondary metrics. Other metrics are described at convenience in the following chapters.

Transactional applications are executed using from 1 to 32 threads. We run several simulations for each benchmark and then compute the average mean. We provide scalability graphic plots to show how applications behave under different number of threads. We also include bar graphs to analyze the performance of many-threaded executions, which are the ones that present more performance pathologies. Bar graphs show the speedup with respect to the reference HTM systems using 32-threaded (16-threaded) executions when studying low-contention (high-contention) applications—we use fewer threads for high-contention applications because they normally do not scale beyond 16 threads.

Modeling power and energy consumption in transactions systems is extremely hard, given that aggressive designs may accelerate execution and thus reuse hardware resources more often. While this behavior may introduce high energy consumption peaks, it may reduce overall execution time. Moreover, the number of instructions may vary depending on the policies implemented in the HTM system, making the energy analysis useless.

Due to the difficulty of providing accurate power/energy metrics, we have decided to present a behavioral analysis instead. Network traffic, point-to-point coherence messages and memory accesses have been characterized. This approach allows us to point out some hints that may help the reader to understand the power and energy implications of the proposed transactional mechanisms.

We evaluate the complexity and area of our proposals qualitatively, comparing the transistors required in our implementations with the hardware cost of state-of-the-art HTM systems.

Chapter 4

A Log-Based Hardware Transactional Memory with Fast Abort Recovery

Data version management (VM) is one of the key aspects of eager[1] HTM systems, and its implementation has a direct impact both in the performance of the system and in the complexity of the hardware design [14].

Late VM systems [5, 33, 97] keep old (pre-transactional) state in the upper levels of the memory hierarchy, buffering the new (speculative) state in private caches [97, 112] or in store buffers [33]. While in late data versioning aborts are fast because it is enough to invalidate those hardware structures that hide the new state, commits require additional data movements. Moreover, late VM systems incur in significant overhead in the case of resource overflow, as they must jump to STM execution [63] or traverse complex hardware structures on cache misses and commits [97].

Early VM [84] systems keep new state in-place in memory, holding the pre-transactional state on a side, commonly a software-resident log [10, 12, 84, 130]. In case of abort, the system must trigger an exception and recover pre-transactional values using a user-level software routine. Nonetheless, commits are immediate—data is already placed in memory. As a result, early VM systems do not suffer from cache/buffer overflows like late VM systems—speculative data can safely be moved across the memory hierarchy.

[1]in this thesis, we refer as eager HTMs those systems that resolve conflicts as soon as they are produced, independently of the VM strategy that they employ.

The obstacles of early and late VM systems have led us to develop FASTM, a log-based HTM system with fast abort recovery. This is an eager HTM system that takes the best of both VM worlds: FASTM keeps both the new state and the pre-transactional state in memory to provide **fast commits and aborts**. FASTM achieves this by pinning down new values in the L1 caches, similar to late VM systems, but with two key differences: (i) transactions update memory in-place, so commit requires no special actions, and (ii) overflows are handled gracefully by using a software-managed log, like in early VM systems.

In FASTM, we change the coherence protocol and the L1 cache controller to guarantee that, if there are no overflows, the old state is in-place in the higher levels of the memory hierarchy. Aborts in FASTM are fast, because they only require the invalidation of the transactional lines that remain in the L1 cache. Nonetheless, since the pre-transactional values are kept in a log on the side, if a transactionally modified line is evicted from the L1 cache the system can recover the old state from the log, using the software abort recovery mechanism—not unlike early VM systems.

This chapter starts presenting a quantitative analysis of the limitations of state-of-the-art eager HTM systems and follows showing a potential study that explains the benefits of having fast abort recovery. Then, it overviews the FASTM system, its underlying hardware and the modifications in the coherence protocol. Afterwards, it summarizes how FASTM executes basic memory operations. Later, it proposes two optimizations for the FASTM system: FASTM-WN and FASTM-SL. The chapter finalizes evaluating FASTM, reviewing related work on unbounded HTM systems and exposing the main conclusions of the study.

4.1 Motivation

Previous studies [26] have claimed that common-case transactions were short and did not usually conflict. However, newer, more complex workloads that are believed to better represent future transactional applications [16] exhibit a significant number of large and/or conflicting transactions. The execution of large transactions has uncovered performance issues with current implementations of both early and late VM systems.

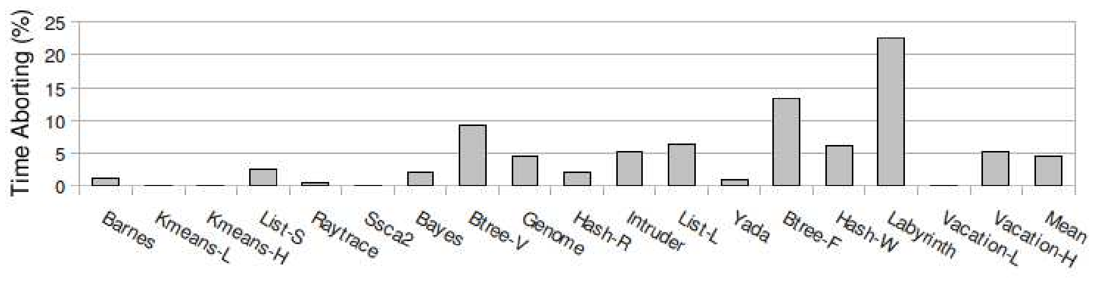


Figure 4.1: Percentage of time spent in abort recovery under 16-threaded LogTM-SE

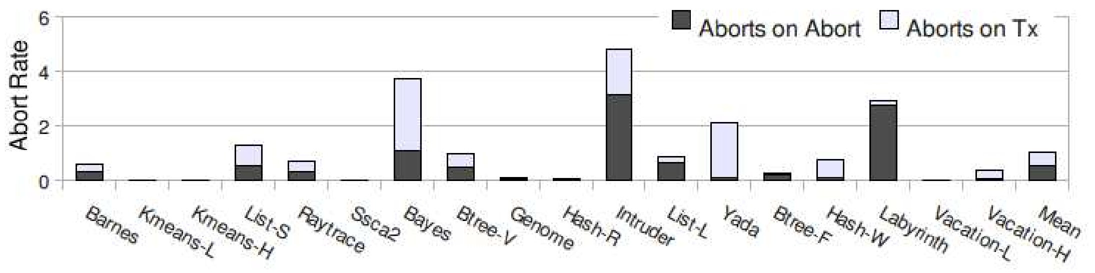


Figure 4.2: Abort rate distribution of 16-threaded LogTM-SE executions

Early VM systems suffer considerable delays when they execute transaction-dominated workloads. Figure 4.1 shows the percentage of time spent in abort recovery after executing 16-threaded TM applications using LogTM-SE, a well-known early VM system. Applications are gathered according to the size of the transactions that they execute; from fine-grained (left) to coarse-grained (right). As it can be seen, TM applications devote substantial part of their time to abort recovery, especially those with huge (*e.g.*, *Hash-write* or *Labyrinth*) or conflicting (*e.g.*, *Btree-fix* or *Intruder*) transactions, due to the overheads of software recovery.

Moreover, slow aborts may exacerbate contention, as many conflicts involve transactions in their abort recovery phase, which in turn provokes more aborts. This happens because an aborting transaction cannot clean (remove the ownership of) its write set until the pre-transactional values are restored. Figure 4.2 classifies the abort rate of 16-threaded LogTM-SE executions in two different groups: those aborts produced by in-flight (non-aborting) transactions (light bar) and those aborts that involve, at least, one transaction that is recovering its old state (dark bar).

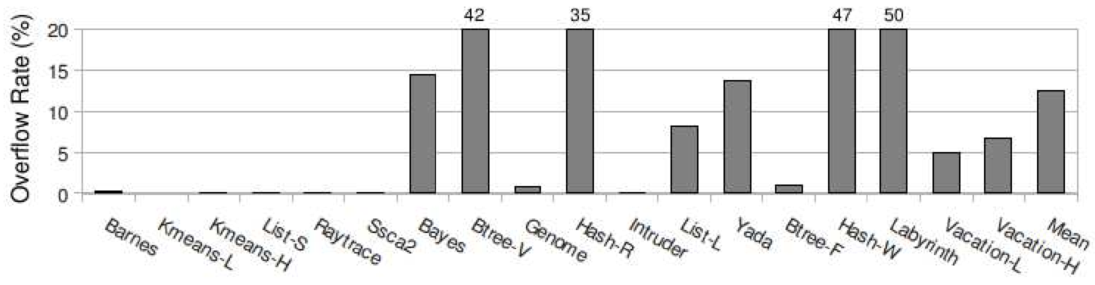


Figure 4.3: Percentage of overflowing transactions in single-threaded LogTM-SE executions

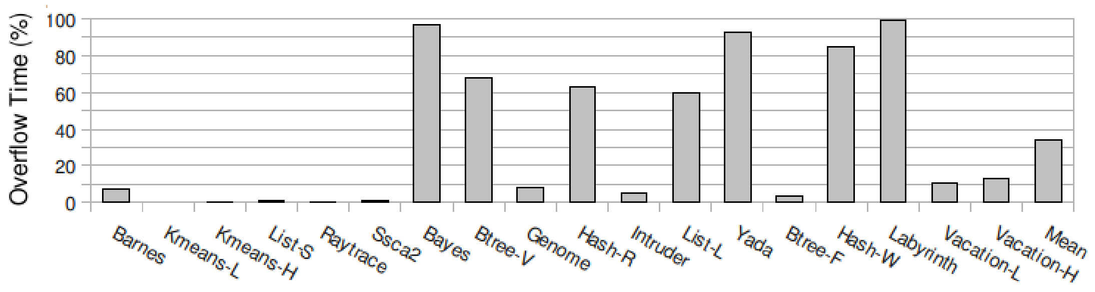


Figure 4.4: Percentage of time spent in overflowing transactions in single-threaded LogTM-SE

As shown in the graph, in high-contention applications (*e.g.*, *List-short*, *Intruder* or *Labyrinth*) an important number of aborts come from aborting transactions.

On the other hand, the overflow mechanism becomes critical in late VM systems. Figure 4.3 shows the percentage of committed transactions that overflow the L1 cache in single-threaded executions of LogTM-SE. The left-sided bars show fine-grained TM executions, while the right-sided bars show coarse-grained TM executions. In contrast to conventional belief, the graph indicates that TM applications (especially those with large transactions, like *Btree-var* or *Hash-read*) commonly evict transactionally written lines from the L1 cache.

Even more important, overflowing transactions cover most of the transactional time in variable- and coarse-grained TM applications. From Figure 4.4, which shows the percentage of execution time (transactional and non-transactional) that LogTM-SE runs an overflowing transaction (single-threaded), we can devise how critical is to not slow down on overflows. However, late VM systems delay those transactions, either executing them in software or walking complex hardware structures on L1 cache misses or on commits. Thus, those systems perform worse

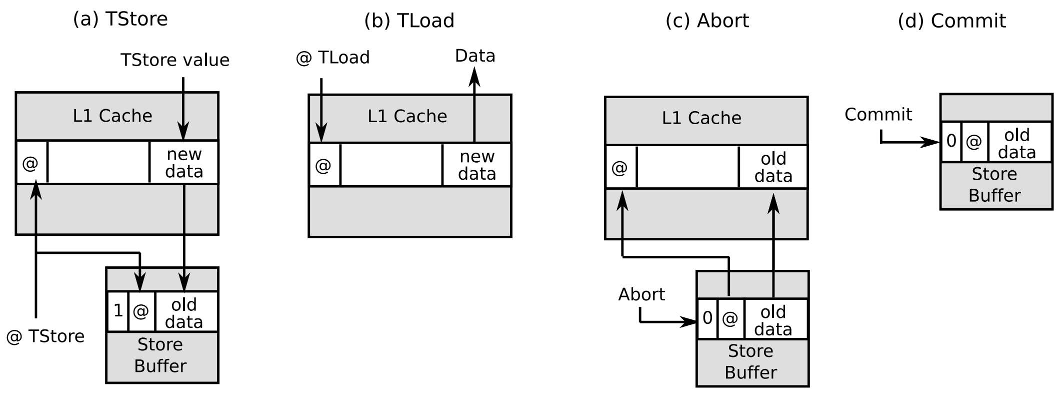


Figure 4.5: Store buffer implementation of an HTM system with early VM

than early VM systems when they execute TM applications populated with large transactions because they (i) retard memory operations, (ii) require additional movements at commit time and (iii) increase the time that conflicting data is exposed to conflicts.

To quantify the overheads associated with data versioning, we analyze the behavior of TM applications on a store-buffered *early* VM system that recovers instantaneously the old state when the aborting transaction fits in the in-core store buffer [70]. Figure 4.5 describes the store-buffer HTM system, which shares some similarities with the Rock HTM implementation [33]. Following is a general picture of how the system interacts with the memory hierarchy when it executes transactional operations.

Transactional Store: A transactional store (TStore) sends the old data from the L1 cache to the store buffer and simultaneously updates the L1 cache with the new data value (Figure 4.5a). At the same time, a CAM search is performed in the buffer using the store address. A match means that this address has been written before in this transaction and the correct (pre-transactional) data is already present in the buffer. If no match is found, the old L1 data is stored in the first free entry of the buffer.

TLoad: A transactional load works identically to the original LogTM-SE proposal [130], getting the (non-)speculative data from the closest level of the memory hierarchy (Figure 4.5b).

Abort: When an abort occurs, the processor is stopped and the state is restored by moving the old values from the buffer to the L1 cache using regular memory write requests (Figure 4.5c). If a line has been evicted from the L1 cache, it is brought from the lower levels of the

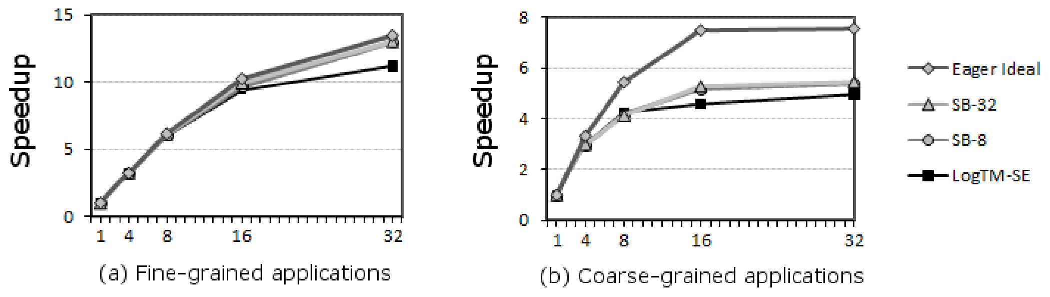


Figure 4.6: Speedup of LogTM-SE, Ideal early VM, Store Buffer (8 entries) and Store Buffer (32 entries) implementations

hierarchy. This is not a problem, because the conflict detection mechanism guarantees that no other processor accesses the line—the abort process must be atomic. In our system, the cache has a single write port, so only one request can be sent at a time. When the abort recovery process finishes, the buffer is cleared.

Commit: On commits, all buffer entries are invalidated by flash-clearing the valid bits (Figure 4.5d). No other action is required.

Overflow: When the buffer overflows, transactions have to be recovered via the software log. In order to avoid unnecessary aborts, this implementation creates the software log always. Hence, transactional stores place the old values in both the store buffer and the software log. On overflow, a special flag is asserted and the store buffer is cleared–from that point, it is not necessary to keep old values in the store buffer. When a transaction aborts, the overflow flag decides if the transaction is recovered via hardware or software.

Figure 4.6 presents the average speedup of LogTM-SE and the store-buffered (SB) HTM implementations normalized to single-threaded LogTM-SE execution. TM applications are grouped according to their granularity. We also present the speedups obtained by an upper-bounded HTM system that spends zero cycles on commits and aborts, which allows us to show the potential of putting into effect an ideal VM mechanism. Two realistic store buffer sizes have been used in this study: store buffers either have 8 or 32 word entries. These sizes correspond to the ones found in commercial processors from the Niagara series [62] and Rock [23], respectively.

As we can see in Figure 4.6, the ideal (zero-cost commit and abort) VM implementation outperforms LogTM-SE by an average 18% on fine-grained applications (32-threads) and approximately by a 70% on variable- and coarse-grained applications (16-threads). Thus, data versioning becomes critical in applications that execute large transactions.

Store-buffered HTM implementations obtain similar performance to the ideal VM mechanism on fine-grained applications, where transactions fit on the store buffer. However, in variable- or coarse-grained applications they behave as LogTM-SE: the ideal performance is downgraded around a 45% (8-entry) and a 44% (32-entry) when store buffers are used for version management. Thus, we can conclude that bigger buffers are needed to achieve close performance to the ideal VM system.

We can draw three main conclusions from the previous study. First, a VM mechanism with fast abort recovery should help to reduce overall execution time and contract the window of conflicts, improving then the scalability of TM applications. Second, such VM mechanism must be able to handle gracefully resource overflows, eliminating large transactions from the critical performance path. Third, this VM mechanism should have enough capacity to accelerate any kind of transaction, given that conventional store buffers are too small to be effective when they are used to maintain transactional values.

4.2 The FasTM System

In FasTM we propose such VM mechanism, which allows us to overcome the major limitations of early and late VM systems. To achieve that goal, FasTM impulses a simple but truly elegant abort strategy: it provides fast abort recovery for short- and medium-sized transactions (common case) and slow software recovery for those transactions that exceed the L1 cache (uncommon case).

4.2.1 FasTM Overview

The novelty in FasTM lies on the way it manages the transactional state and in its abort recovery mechanism. Following the example of many late VM systems [97, 112], FasTM utilizes a new coherence protocol for the L1 cache—we call it Transactional Cache Coherence Protocol (TMESI for short).

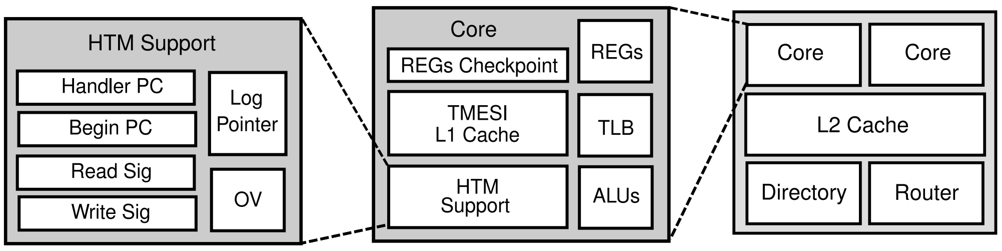


Figure 4.7: Hardware support for FASTM

TMESI is a write-back protocol that provides fast commits because it does not hide transactional updates from the memory hierarchy—this is the main reason why FASTM is taking for an *early* VM system—but it does enforce the following condition: transactionally modified lines are "pinned" in the L1 cache (they cannot write back) to guarantee that a valid copy of the pre-transactional version of the line exists in the memory hierarchy until commit/abort time (or until an overflow occurs). This operation is similar to some Thread-Level Speculation protocols [115]. Section 4.3 presents a throughout description of the TMESI protocol.

With the TMESI protocol, the system guarantees that if no overflow occurs, the old values are still in-place in the higher levels of the memory hierarchy. Hence, if a transaction that has not overflowed the L1 cache aborts, FASTM provides a very fast abort mechanism: it simply invalidates the lines modified by the transaction (this is a silent invalidation, more on this later).

However, if an overflowed transaction aborts, FASTM falls back to a software recovery mechanism similar to that employed in LogTM-SE [130]. The software abort recovery process requires just a few registers to hold the last entry of the log, the address of the abort recovery routine and the Program Counter (PC) of the current transaction.

Like most of the early VM implementations, FASTM performs eager conflict detection and eager conflict resolution. FASTM borrows the conflict detection engine from LogTM-SE, where the directory forwards transactional requests to the private caches. Moreover, as it is described in Section 4.7.4, FASTM supports multiple conflict resolution policies.

4.2.2 Hardware Support

FASTM is an eager HTM system based on LogTM-SE [130], so our proposal requires mostly the same hardware support. (A scheme of FASTM's hardware support is shown in Figure 4.7.) FASTM uses two hardware signatures to track transactional accesses: a Read Signature to identify read conflicts and a Write Signature to detect write conflicts in case of overflow.

Also, it keeps a software log in the same way as log-based HTM systems do: each transactional store copies the old value to the log before updating the memory with the new value. We assume that logging is a dual-phase process where, (1) the old line is brought to the processor and is written in the first free entry of the log and (2) the new value is stored in the cache. The combination of the signatures and the software log allows FASTM to gracefully handle cache overflows.

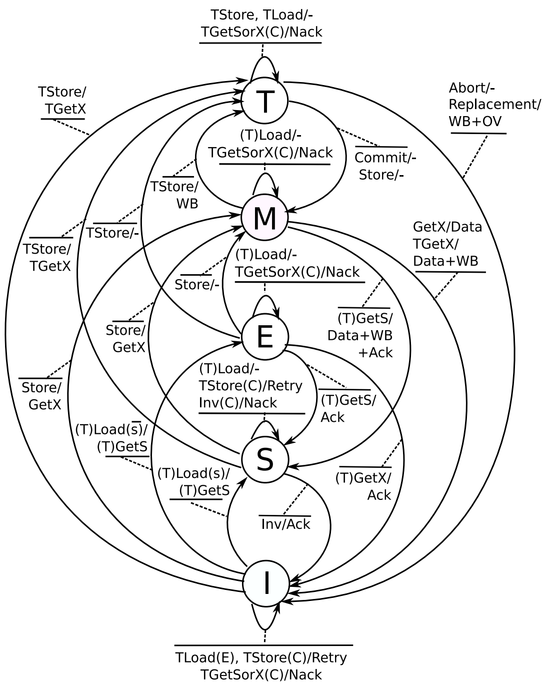
To support eager conflict management, FASTM keeps evicted L1 data in a special directory state, which maintains the last owner (sharers) of the line. Hence, future requests to that line are forwarded to the last owner (sharers), who validates the consistency of the transaction by clashing the requested address into the Read or the Write Signature. Each core also incorporates an OV bit that is set when a transaction replaces a transactionally written line.

4.3 The Transactional L1 Cache Coherence Protocol

In order to allow a special handling of transactional stores, TMESI modifies the classical MESI protocol to put said lines to a new state, named T, where they persist until the transaction commits, aborts or overflows. Therefore, FASTM requires, as it is shown in Figure 4.8, an extra bit to encode the T state and some logic to identify transactional stores.

These T lines are also used to detect conflicts among transactions, so the Write Signature only contains the addresses of lines that overflow (get evicted from) the L1 cache. This fact reduces the aliasing in the Write Signature, increasing its fidelity.

Figure 4.8 shows the principal state transitions of the TMESI coherence protocol. In the diagram, the triggering message is written before the slash and its associated action after the slash (‘–’ means none). *TStore* and *TLoad* are memory accesses produced inside a transaction. *GetS* is a directory forwarding load request from a remote processor, *GetX* is a forwarding transactional write request. Transactional requests are identified by adding a ‘T’ prefix before the requested message (*TGetS* for *TLoads*, *TGetX* for *TStores*).



State	Trans (T)	Dirty (D)	Valid (V)
T	1	1	0
M	0	1	0
E	0	1	1
S	0	0	1
I	0	0	0

Response Message	Description
Ack Nack	Acknowledgement Negative Acknowledgement

Response Actions	Description
OV WB Retry	Notify Overflow L2 Cache Write-Back Re-Issue the Conflicting Request

Local/Request Message	Description
TLoad, TStore	Transactional (Tx) Load, Tx Store
TGetS, TGetX	Forwarded Tx Read (Write) Request
(T)GetS, (T)GetX	Either Tx or Non-Tx Request
TGetS(C), TGetX(C)	Conflicting Read (Write) Request
TGetSorX	Either Read or Write Tx Request
Commit, Abort	Local Commit (Abort) Signal

Figure 4.8: TMESI coherence protocol transitions

If a forwarded message generates a conflict (*TGetS(C)* or *TGetX(C)* in the diagram), the requested line remains in the same state, sending a *Nack* message to the requester. Then, requester can retry the memory access or abort the transaction. The `WB` action pushes the line to the higher levels of the memory hierarchy. *Replacement* indicates an L1 cache eviction. In the case of replacing a transactional line, a set of overflowed actions are required (`OV` actions). `Commit` and `Abort` actions also trigger transitions to the Modified or Invalid states. A detailed explanation of the TMESI transitions is presented in Section 4.4.

We want to note that the TMESI protocol only works for CMP systems with single-threaded cores. In a simultaneous multi-threaded (SMT) environment, it is necessary to prevent accesses to `T`-state lines to those threads that share the core with the owner of the line. There are two different solutions to implement FASTM on SMT systems: (a) keep always written locations in the Write Signature and enforce signature matching even in the case of hitting a `T` line or (b) add a *thread id* on each entry of the cache that informs who is the current owner of a transactionally written line.

4.4 FASTM Transactional Operations

This section describes how FASTM operates, explaining in detail how transactional lines interact with the system. We present the basic operations of the system and describe L1 cache replacements and the mechanism for abort recovery. For our discussion we will assume a CMP system with single-threaded cores and two levels of caches, where the L1 cache is private per core and the L2 cache is shared. Coherency is implemented using a directory at the L2 cache.

4.4.1 Transactional Stores

To understand how transactional stores operate, assume a core C_0 that performs a *TStore* instruction. If C_0 has the line in its L1 cache in an exclusive (T or E) state, it changes the cache state to T and the *TStore* completes immediately. If the line was previously written by C_0 inside a transaction that has already committed, or by non-transactional code, the line may be in C_0's L1 cache in the M state. If so, then C_0 must write back the line data to the L2 cache before transitioning the line to the T state and completing the *TStore*. This write-back does not generate any coherence requests to the other L1 caches, but it is necessary for guaranteeing that the L2 cache always has the correct pre-transactional state.

In Figure 4.9a we can see an example of the case where C_0 misses in the L1 cache (having the line in the S state is identical). The left (right) of the figure shows the state of the system before (after) the *TStore*. When C_0 misses in its L1 cache (step 1), it requests the line from the directory (step 2), and the directory forwards the request (*TGetX*) to the line current owner, in this case C_1 (step 3). If C_1 has the line in the T state, it *nacks* the request directly without checking the signatures. Otherwise, it checks its Read and Write signatures to detect conflicts with the requesting transaction (step 4). If a positive match is found, C_1 *nacks* the request from C_0 and the conflict resolution mechanism kicks in. If the line is not being accessed by any transaction (*i.e.*, C_1 has it in M, or E state), the directory gives the ownership to C_0, invalidating all other copies of the line (in this case C_1).

With MESI, if C_1 has the line in M state it must forward the line data to C_0 before C_0 can become the new owner of the line. In TMESI, the L2 cache must always have a copy of the old value in order to guarantee correct abort recovery for non-overflowing transactions. For this reason, C_1 also sends a copy of the forwarded line to the L2 cache (step 5) before relinquishing

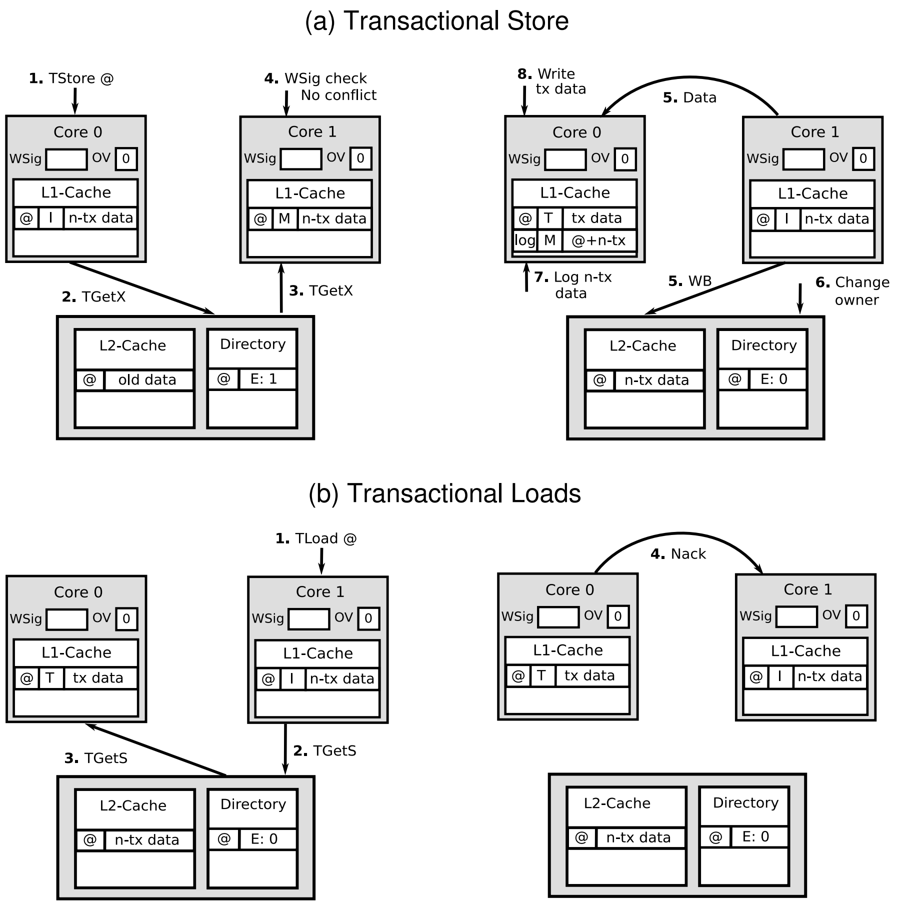


Figure 4.9: Non-conflicting TStore (a) and conflicting TLoad (b) in FASTM

ownership of the line to $\mathtt{C}_0$ (step 6) and allowing $\mathtt{C}_0$ to safely write the transactional value (step 7 and 8).

4.4.2 Transactional Loads

Now, assume a core $\mathtt{C}_1$ that performs a transactional read (*TLoad*) operation. In FASTM, *TLoads* are performed as regular loads. However, in order to maintain transactional coherency, the *TLoad* address must be added to the Read Signature of $\mathtt{C}_1$, which is used to detect conflicts with remote transactional stores. $\mathtt{C}_1$ only has to check for conflicts when loading a line that is not present in its L1 cache. In this case, $\mathtt{C}_1$ must request the line from the directory in the L2 cache, which serves the line if there are no writers. If there is a writer, the directory forwards the request to the core that owns the line.

In Figure 4.9b we can see how the previous example follows in the case where core C_1 tries to read a transactional written line by core C_0. The left (right) of the figure shows the state of the system before (after) the *TLoad*. In this context, C_1 introduces a *TLoad* operation that misses the L1 cache (the line is in I state, step 1). When C_0 receives the forwarding read request (*TGetS*, step 2 and 3), C_0 must acknowledge it. As C_0 has the line in its L1 cache in T state (*i.e.*, it is a transactional, non-oveflowed line), it sends a *Nack* reply to C_1 and the conflict is resolved according to the conflict resolution policy (step 4).

If the requested line is not in T state or it is not in C_1's L1 cache, then C_1 must check its Write Signature. This is necessary to guarantee coherence for transactions that overflow the L1 cache. If there is a match in the signature, C_1 replies to C_0 with a *Nack* message. Otherwise, the line moves to the S state and, if the line was previously in the M state, C_1 forwards the data to C_0 and also writes it back to the L2 cache (this is the same as in a typical MESI).

4.4.3 Transactional Cache Replacements

Figure 4.10a assumes a core C_0 that replaces a line with transactional modifications. In FASTM evictions of lines in T state write back the speculative values to the higher levels of the memory hierarchy, similar to evictions of lines in M state. This is analogous to other log-based HTM systems (those that implement early VM), and it is safe to do because the pre-transactional values are kept in a software log. Nonetheless, the system must perform some actions before pushing the speculative data to the L2 cache.

First, the evicted line address must be added to C_0's Write Signature. Note that in FASTM non-overflowed written cache lines do not insert their addresses in the Write Signature, which allows the system to track the write set in a more accurate precision. The directory maintains as the owner of the line the current core (C_0) and will forward all future remote requests to it. As discussed earlier, upon receiving a remote request C_0 will check its Read and/or Write signatures to discover conflicts. If the C_0 evicted line is also replaced from the L2 cache, the request is forwarded to all the processors, which must check their signatures. This fact permits the conflict detection engine to identify collisions that involve evicted transactionally written lines.

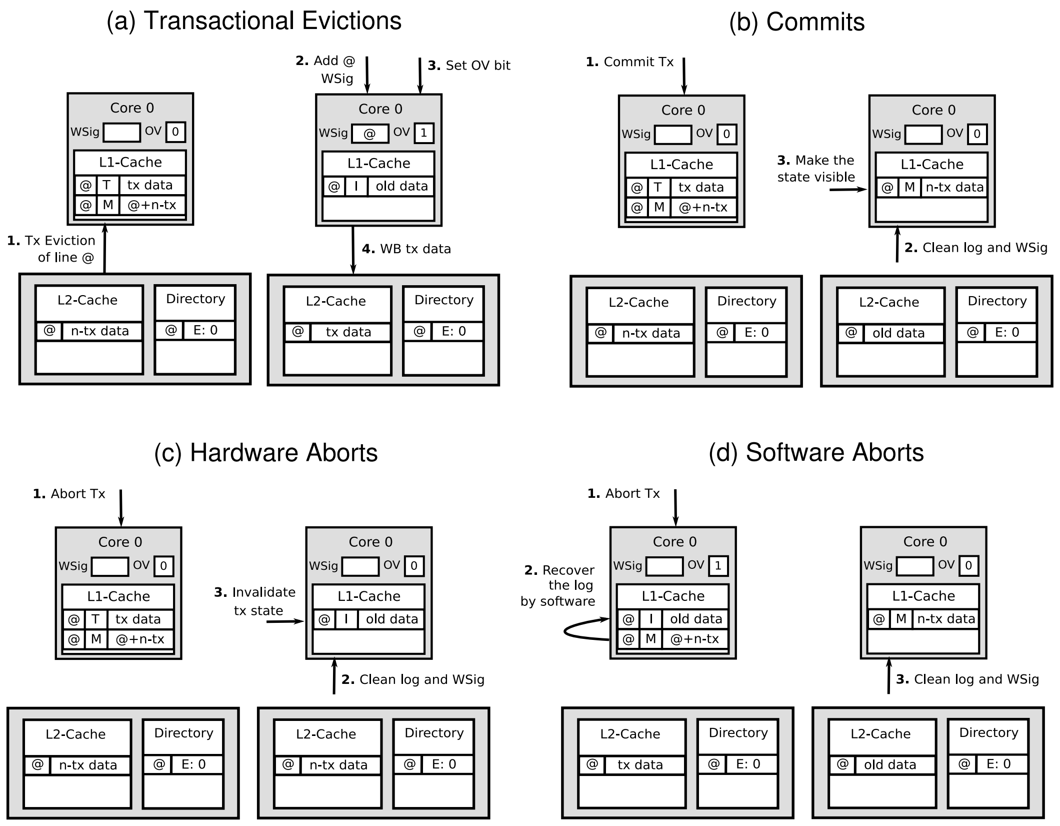


Figure 4.10: Transactional replacements (a), commits (b) and aborts (c and d) in FASTM

Second, a transaction overflow flag in C_0 is asserted to inform the processor that the transaction has to be aborted by software. In FASTM, we have chosen to write all the updated lines in the software log—this fact allows software abort recovery at any time.

An alternative is to only insert overflowed lines in the software log (instead of all updated lines). This approach is more efficient, because it reduces abort recovery time of overflowed transactions, given that fewer lines must be restored by the software routine. Moreover, this results to less cache pollution (the software log is in cacheable memory) which may result in fewer transactional evictions. We propose such optimization in Section 4.6.

4.4.4 Committing Transactions

FASTM provides, like other early VM systems, a fast commit even for overflowed transactions. FASTM only commits consistent transactions, therefore no additional actions are needed to validate the speculative state. In FASTM, a committing transaction first flush-clears the T bit of all cache lines, moving all T lines to M, and then releases the signatures. Figure 4.10b shows how FASTM makes visible the transactional state at commit time.

Notice that non-cached lines do not require any commit action, because transactional modifications are already in the memory hierarchy. In contrast to late VM schemes [97, 124], our system does not require sending state updates to the directory. Instead, the directory already has the committer as the owner of the line (it acquired ownership during the execution of the transaction).

4.4.5 Aborting Transactions

FASTM uses a hardware-accelerated abort recovery mechanism for non-overflowed transactions (Figure 4.10c) and a software abort recovery mechanism for transactions that have evicted lines in the T state (Figure 4.10d). The processor decides which of the two recovery mechanisms applies by checking its overflow (OV) flag.

Non-overflowed transactions use the coherence protocol to discard transactional modifications. This process is performed by silently invalidating all the T state lines in the L1 cache (the directory is updated lazily by future requests). Hence, when the transaction restarts again, it must re-acquire the ownership of each line. This can be safely done because the L2 cache keeps the pre-transactional state.

Assume that core C_0 aborts and now core C_1 requests a line that C_0 wrote inside the aborted transaction. First, the directory will forward the request to C_0 since it is still the owner. C_0 acknowledges the request, informing C_1 that it (C_0) is no longer the owner. Then, C_1 will take the line from the L2 instead, which still keeps the pre-transactional value, and the directory will be updated. This lazy directory update removes unnecessary communication with shared resources at abort time, allowing a fast abort recovery.

The invalidation of T state lines increases the number of L1 misses on restarted transactions. However, this situation is not critical mainly for two reasons. First, most transactions have considerably smaller write sets than read sets, so the rate of L1 store misses is not a bottleneck (read lines are not invalidated in FASTM). Second, these L1 misses are served faster than conventional L1 misses because these lines are still owned by the aborted transaction.

Let’s assume now that core C_0 aborts and tries to re-acquire a line invalidated by the fast abort mechanism. C_0 requests the line from the directory, which still has C_0 as its owner. Thus, the line can be directly served from the L2 cache, without requiring coherency operations or signature checking.

Transactions that overflow the L1 cache are recovered by software by taking a trap to the recovery handler. The recovery handler is a software routine that walks the log in reverse order and, for each entry, writes the logged data to its corresponding place in memory. Notice that some of the `T` state lines may be overwritten by the recovery handler. Such writes are performed by non-transactional stores, moving the lines from `T` to `M`. When the software abort-recovery mechanism finishes, it returns control to the hardware.

Both the hardware and the software mechanisms release the signatures when the recovery process finishes.

4.5 FasTM with Wake-up Notification

FasTM follows the same conflict resolution policy than other high-performance eager HTM systems like LogTM-SE [130], MetaTM [98] or the eager mode of FlexTM [111]: it stalls on a conflict and then retries the non-completed memory request until the inconsistency disappears—*i.e.*, the conflicting transaction commits or aborts. While this strategy permits the conservation of useful transactional work, it also introduces unnecessary traffic on the on-chip network, which increments the energy consumed by the application and saturates shared resources (*e.g.*, the network and the directory modules), slowing down conflict-free transactions.

In this section, we propose a novel technique to manage stalled transactions (we called it *wake-up notification*). Instead of continuously retrying those memory accesses that conflict, we stop polling on those cores that cannot complete a memory operation. These cores remain disabled (they do not schedule any operation to the memory subsystem) until the conflicting transactions either commit or abort. At that moment, the system enables the core and the transaction restarts its execution from the conflicting point.

4.5.1 Conflict Tracking

Wake-up notification extends FASTM cores with one bit vector, called Wake-up List (WL for short) and an integer counter, called Nacks. The WL tracks those cores that are executing a transaction which has requested a location being read or written by the in-flight transaction. Thus, this vector has a bit per on-chip core, which is set each time a conflict is notified. In other words, the WL maintains a list of the cores stopped by the current transaction.

The Nacks counter, on the other hand, maintains the number of remote, non-committed transactions that conflict with the transaction executed in the non-operative core. Thus, when the counter is positive, it means that the conflict is still present in the system, while when the counter is zero, the core should be enabled.

4.5.2 The Wake-up Mechanism

The wake-up mechanism notifies stalled transactions that the conflict who caused the disablement of the core has disappeared. This action, which is performed by a committing transaction before it makes the state visible or by an aborting transaction before releasing the read and write sets, consists on sending a *WakeUp* message to all the cores present in the `WL` list—those cores stalled by the committer/aborter.

When a core receives a *WakeUp* message, it is because a remote conflicting transaction has finished its execution. However, it may be the case that other transactions still own the requested data, making useless the activation of the core. Instead of starting polling again, the receiver decrements the Nacks counter and acknowledges the request. The core only re-schedules the offending memory request when the Nacks counter is zero—*i.e.*, all the transactions that participated in the original conflict have committed or aborted. Of course, the conflict may remain, as younger transaction may have acquired the permissions of the line while the core was disabled.

We want to note that the use of wake-up notification is independent of the conflict resolution policy used on the HTM system. Our mechanism does not say *which* transaction has to be stalled nor detects cycles among stalled transactions—there are plenty of examples in the literature that explain how to deal with that. Instead, we focus on *how* to handle contending transactions efficiently, minimizing the impact of retrying redundant memory instructions. Moreover, this technique can be applied to any eager HTM system, without mattering its VM mechanism.

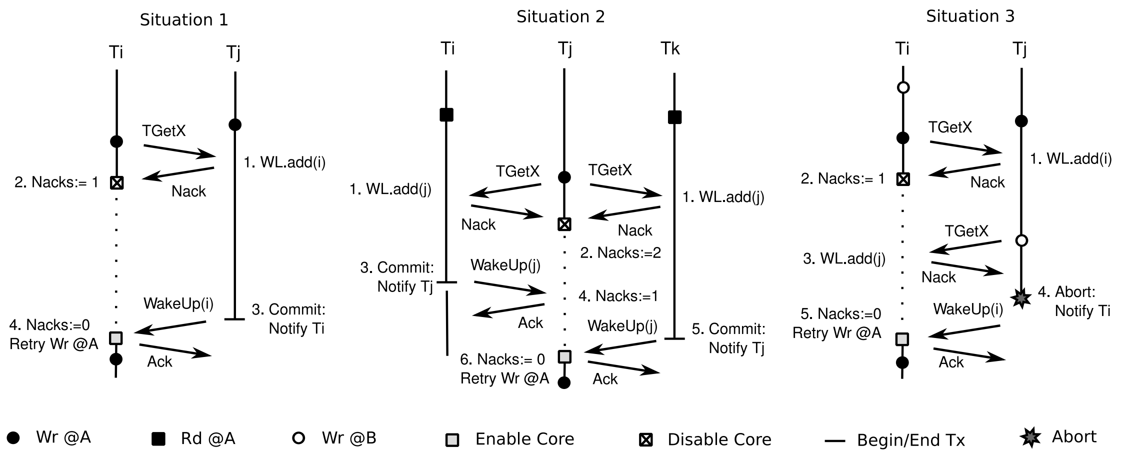


Figure 4.11: Examples of the wake-up notification mechanism

4.5.3 FASTM-WN: Examples of Wake-up Notification

Figure 4.11 shows how FASTM integrates wake-up notification in different situations, building the FASTM-WN system. In Situation 1, transaction `Ti` asks for a line that belongs to transaction `Tj`. When `Tj` receives the petition, it updates its WL list, adding core `Ci` as a conflicting core, and replies `Ti` with a *Nack* message. When core `Ci` catches the message, it updates the Nacks counter setting it to one and stops polling the conflicting line. Eventually, transaction `Tj` commits, sending a *WakeUp* message to those cores marked in its WL (in the case, `Ti`). As a result, the system awakes core `Ci` (Nacks value is decremented to zero) and it resumes the execution of `Ti`.

Situation 2 shows a similar scenario, but in this case transactions `Ti` and `Tk` own the line. Thus, when `Tj` tries to acquire the data, it fails in its attempt, setting the Nacks counter at 2. Thus, `Tj` must wait until `Ti` (first) and `Tk` (later) commit to retry the execution of the stalled transaction. In Situation 3, there is a crossed conflict between transactions `Ti` and `Tj`. In order to guarantee forward progress, the aborting core `Cj` has to send a *WakeUp* message to revive core `Ci`, who retries the original conflicting request.

It can be the case that, due to external effects (a gamma ray flips a bit in a *Wakeup* message while it is crossing the network), a wake-up notification does not arrive to the consumer. To avoid deadlocks—transactions that are never revived by the *committing* core—FASTM-WN couples the system with a mechanism that is triggered if a timeout threshold is reached.

This mechanism aborts the stopped transaction and re-executes it afterwards to guarantee forward progress in the application.

4.6 FASTM with Selective Logging

Log-based HTM systems (FASTM included) must face three main challenges that may slow down transactional execution. First, writing pre-transactional values to the log always before a transactional store enlarges the latency of those memory accesses—speculative values cannot be written in memory until the old data is logged. Second, the software log is maintained in cacheable memory, which reduces the buffering capacity of the transactional caches—in other words, the logging mechanism increases the possibilities of evicting transactional data. Third, in log-based HTM systems *all* transactionally written lines are placed in the log, even if the speculative data fits in the L1 cache. Thus, if an overflowing transaction aborts, it has to restore the whole log using a *slow* software routine, ignoring the built-in hardware support of transactional caches [5, 12, 71, 97].

In order to address the above issues, we propose *selective logging*, a novel VM technique that only logs the pre-transactional values of those memory blocks that the hardware cannot recover—*e.g.*, a non-committed speculative write that overflows the transactional L1 cache. Hence, he idea behind selective logging is rather simple but effective. By adding a few additional hardware steps on resource overflows (uncommon event), we are able to (i) accelerate most of the memory updates within a transaction, (ii) reduce the size of the software log, which diminishes the L1 cache pollution and (iii) accelerate the abort recovery process because fewer lines must be restored by software.

Selective logging can easily be included in the FASTM framework to build a powerful eager HTM system that we called FASTM-SL. The following sections describe the selective logging mechanism, the hardware/software support that it requires and the FASTM-SL system.

4.6.1 The Selective Logging Mechanism

When an HTM system incorporates support for selective logging, transactional stores do not carry additional actions—*i.e.*, it is not necessary to write in the log the old state before updating the memory. However, when a transactional line is evicted from the L1 cache, the processor stops conventional execution (the memory instruction that generates the cache miss remains incomplete) and starts executing a microcode routine. This firmware loads the old value of the line from the L2 cache into a special register, and stores the old data and the corresponding memory address in the first free entry of the software log. After that, the processor re-schedules the memory instruction that produced the cache replacement and continues executing the transaction.

Like other log-based HTM systems, commits do not require additional actions, given that the transactional state can harmlessly flow though the memory hierarchy. Nonetheless, when an overflowing transaction aborts it has to perform a two-phase procedure. First, the hardware invalidates all the transactional lines in the L1 cache, clearing the transactional state from caches. Then, the processor throws an exception and traps to the user (or system) software layer, which undoes the modifications introduced by the transaction.

4.6.2 Pushing Physical Addresses in the Log

By deferring log updates to L1 eviction time, selective logging requires a subtle modification in the way the log is stored in memory. More specifically, traditional log-based HTM systems use *logical* addresses to track the location of pre-transactional data. Logical addresses are readily available at the transactional store issue time (*i.e.*, the time the log info is collected). The benefit of using logical addresses in the log is that the software recovery routine can be done in user-space. In selective logging, on the other hand, the system collects the log info at the time an L1 cache line is evicted. At this point, logical addresses are not available (most memory systems use physical addresses), but using physical addresses in the log poses a security risk though.

In order to address the above issues, we propose to move the transaction abort recovery handler in the Operating System (OS). In this case, when an overflowing transaction aborts, the hardware raises an exception that calls the OS abort recovery routine. The OS recovers the log using physical addresses and returns control to the application. Note that logical-to-physical translation is not needed when the OS is undoing the log—the TLB is automatically bypassed. Moreover, the actual log memory must be only visible to the OS, otherwise user applications can reverse-engineer the logical-to-physical memory mapping. This requires that transactional applications execute a log creation system call at init time. The memory of the log is thus kept in OS memory, and is hidden from the application.

Some HTM systems may not want to expose the abort recovery to the OS. An alternative consists on obtaining the evicted virtual[2] address of the memory block from somewhere, so the system can recover the pre-transactional state in the user-space. A simple way to do it resides on modifying the TLB to provide *reverse* translation from physical to virtual page memory addresses. Another design option consists on using a virtually-tagged L1 data cache [2, 39, 56]. ARM v5 [56] is just an example of a commercial processor that uses virtually-tagged caches. Discussing the viability of virtually-tagged caches or reverse TLB translation is out of the scope of this thesis (we refer the reader to [19]).

4.6.3 FASTM-SL: Adding Selective Logging to FASTM

This sections describes how the FASTM infrastructure can be extended to support selective logging—we call this system FASTM-SL. FASTM-SL differs from FASTM in the way it updates the software log and how it recovers the pre-transactional state when aborting an overflowing transaction. While FASTM logs the values of *all* transactional stores (at least the first time they write a line inside a transaction), FASTM-SL only logs the values of transactional evicted data. Thus, if an overflowing transaction aborts, FASTM has to restore the entire pre-transactional state by software. Instead, FASTM-SL can take advantage from the innate in-cache support for clearing non-evicted cache lines.

For FASTM-SL, we assume that the log contains physical addresses, and thus it has to be recovered in privileged mode. When a transactionally written line is evicted from the L1 cache, FASTM-SL has to construct a new log entry. We use the example of Figure 4.12 to describe how the selective logging machinery handles the eviction of a T-state line (step 1). First, the eviction process is put on hold, and the core sends a request to the L2 cache for the previous version of the line (step 2).

The requested data, together with the physical address of the line are temporarily stored in a special register. At this point, the data in the special register is written to the first free entry

[2] we assume that logical addresses are analogous to virtual addresses, as most 64-bit architectures (*e.g.*, Intel x86) turn off segmentation. In fact, the majority of OS Kernels omit segmentation when possible. Systems that require segmentation can trivially add a step in the translation to obtain the logical address from the virtual address.

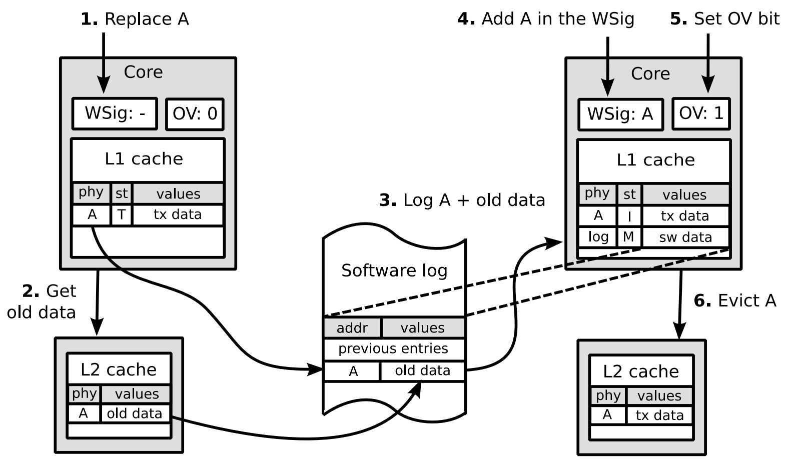


Figure 4.12: L1 Cache replacement actions in FASTM-SL

of the log using regular (*i.e.*, non-transactional) memory operations (step 3). Then, the physical address of the line is added to the Write Signature (step 4) and the OV bit is set (step 5). Finally, the transactional line is evicted to the L2 cache (step 6). This process maintains the atomicity of the store operation, a requirement for an in-order core. However, non-blocking out-of-order processors can be overlap the logging process with other computation.

When an overflowed transaction aborts, FASTM-SL has to restore the values modified during its execution. For non-evicted data it is enough to invalidate T-state lines (by flash-clearing the state bits), as pre-transactional values are still valid in the L2 cache. However, transactional replaced data has to be restored by software because the L2 cache does not hold the old state anymore. Hence, the system triggers an exception, which jumps to an Operating System routine that walks the log in reverse order to undo the changes introduced by the aborted transaction.

Note that, in contrast to FASTM, FASTM-SL *only* has to restore those lines that have been evicted from the L1 cache during the in-flight transaction—those lines that fit in the L1 cache are invalidated by the underlying hardware and, eventually, the core will obtain the valid data from the L2 cache using conventional coherence requests when the transaction restarts. As a result, the size of the log (and thus the time spent in software abort recovery) is reduced considerably.

4.6.4 Discussion

Selective logging introduces some complexity in the FASTM system. In fine-grained applications overflows are not common, so FASTM does not experiment long delays. Moreover, accesses to the software log experiment high locality, and thus writing the log takes small overhead—most of the memory accesses hit the L1 cache. In addition to this, few transactions overflow the L1 cache (and almost none of them abort), so the size of the software log is not so hazardous.

Having said that, someone could say that the amount of complexity of the technique—it may require changes in the hardware or in the OS—is too high and it does not worth the effort. However, we see reasons to not believe such thing. First, there is a set of TM applications—those that execute large transactions, evaluated in Section 4.7.3—that can take advantage of selective logging. It is not clear if those applications will be the norm in the near future, but some people claim that they will [16]. Hence, it is important to have this thought in mind when designing HTM systems, and selective logging certainly helps those coarse-grained applications.

Second, the efficient log mechanism considered in log-based HTM systems may be too expensive for next generation of commercial processors, which may decide to implement cheaper but slower logging strategies. In this situations, selective logging avoids the overhead of accessing the log on each transactional store; in fact it removes the cost of logging if the transaction fits in the L1 cache. Third, selective logging opens new avenues for HTM systems that do not support unbounded transactions (more on this in Section 5.3).

However, the hybrid recovery solution complicates the abort mechanism, which must maintain the atomicity of a dual phase hardware/software abort. The upside of maintaining the software abort for all updated lines is that it allows the use of mechanisms like those of LogTM-VSE [119] to survive context switches or page faults. Thus, when this happens in FASTM-SL, we opt to abort the in-flight transaction and restart it in the original FASTM mode.

4.7 Evaluation

For our analysis we have chosen to compare FASTM with two eager HTM systems that implement early VM, although with different underlying mechanisms. The first one, which serves as our baseline, is LogTM-SE, particularly the implementation that is distributed with GEMS 2.0 [80]. The second one, is an idealized early VM system with zero-cost abort recovery that servers as our upper-bound.

We have used the **Stall** conflict resolution policy for the comparisons between FASTM and the other HTM systems. Stall is the policy implemented by LogTM-SE [130]. After detecting a conflict between two transactions, this policy stalls the requester, who waits until the other transaction commits. However, to avoid cyclical dependences among stalled transactions, transactions must inform a centralized cycle-detector when they are stalled. If a dependence cycle occurs, a timestamp determines the younger transaction that participates in the cycle and aborts it. After recovery, an exponential backoff is performed to guarantee progress.

We decided to use the Stall conflict resolution policy for all the comparisons between LogTM-SE and FASTM for two main reasons. First, this policy minimizes the number of aborts, which become critical in an HTM with software abort recovery (also, by minimizing aborts we are conservative in how much FASTM improves over LogTM-SE). Second, by using the Stall policy for our evaluation it is easier to compare FASTM results with previous LogTM-SE characterizations [70, 106, 121]. In Section 4.7.4 we describe other conflict resolution policies and we discuss about how they behave in LogTM-SE/FASTM.

Moreover, we have also evaluated **FASTM-SIG**, a variation of FASTM where all *TStore* addresses are added to the Write signature (remember that FASTM only updates the Write signature with T state lines that get evicted from the L1 cache). Studying this alternative allows us to determine the performance benefits of reducing aliasing in the signatures.

In addition to the above proposals, we have also evaluate the two optimizations for FASTM presented in this thesis: **FASTM-WN** and **FASTM-SL**. FASTM-WN implements wake-up notification to save energy and bandwidth on conflicting memory accesses. FASTM-SL incorporates the selective logging mechanism in hardware exposing physical addresses to the log.

4.7.1 FASTM Performance Analysis

Figure 4.13 presents the time distribution of LogTM-SE (labeled L), FASTM (labeled F) and Ideal (labeled I) HTM systems in their 32-threaded executions (for low-contention applica-

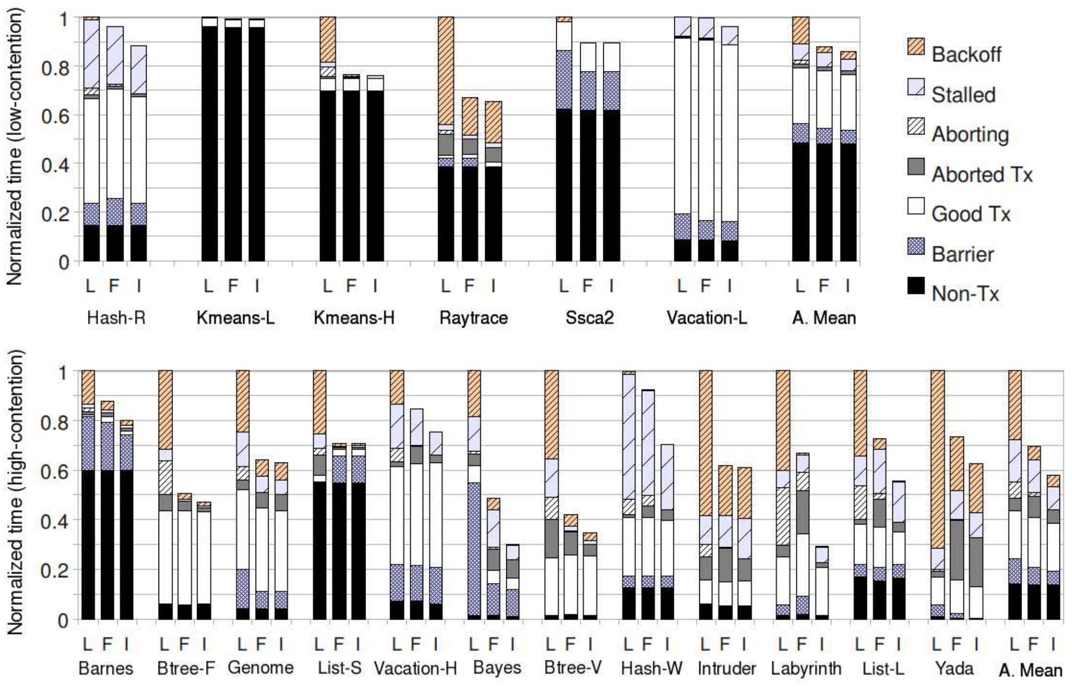


Figure 4.13: Distributed execution time of low-contention (top, 32 threads) and medium- and high-contention (bottom, 16 threads) TM applications under LogTM-SE (L), FASTM (F) and Ideal (I) HTM systems

tions, top of Figure 4.13) and in their 16-threaded executions (for high-contention[3] applications, bottom of Figure 4.13) using the Stall conflict resolution policy.

The execution time has been normalized to the 32-threaded (low-contention) and 16-threaded (high-contention) LogTM-SE execution and is broken down to: non-transactional and barrier cycles (labeled Non-Tx and Barrier), the time spent in committed transactions (labeled Good Tx), the time that is wasted in non-useful work discarded from aborted transactions (labeled Aborted Tx), the time spent in abort recovery (labeled Aborting), the time that transactions remain stalled waiting for a conflict to be resolved (labeled Stalled), and the time that processors execute the exponential backoff after aborting (labeled Backoff).

As it can be seen in Figure 4.13, FASTM has an average speedup of 16% (low-contention) and 44% (high-contention) over LogTM-SE, achieving, in most of the workloads, similar performance to the ideal VM approach, which uses a zero-cycle abort recovery mechanism and

[3]we refer as high-contention applications those that presented medium and high abort rates in Section 3.3.2

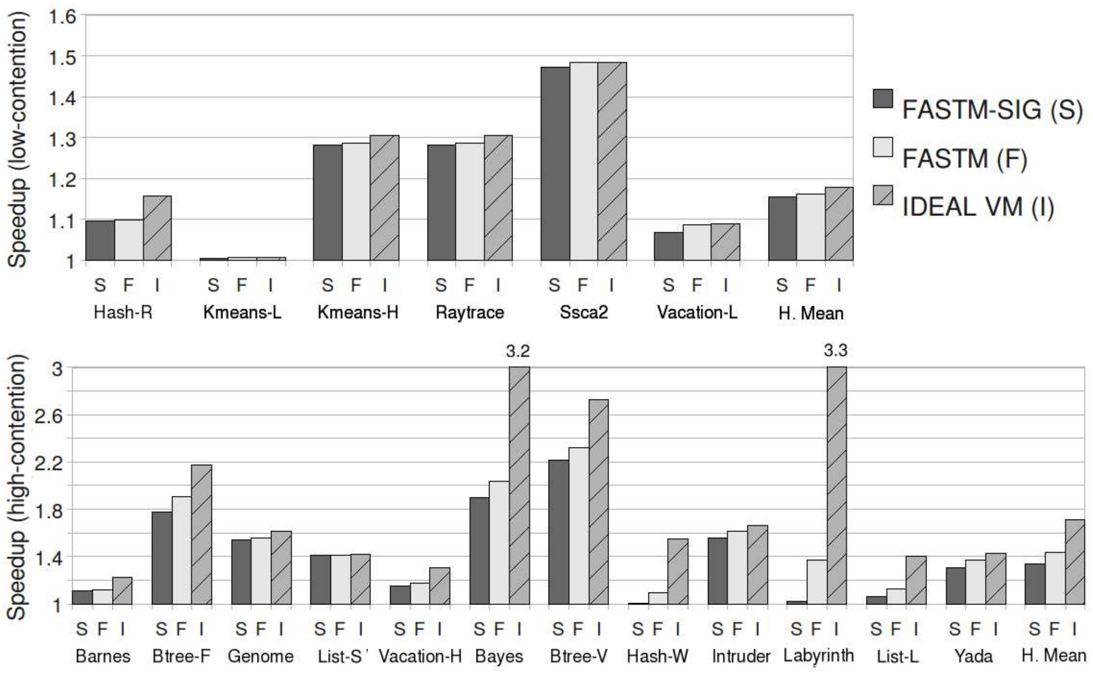


Figure 4.14: Performance improvement of FASTM-SIG (S), FASTM (F) and Ideal VM (I) HTM systems over LogTM-SE in low-contention (top, 32 threads) and medium- and high-contention (bottom, 16 threads) TM applications

perfect signatures. The benefit is especially notable in some coarse-grained applications like *Bayes* or *Btree-fix*, where FASTM obtains more than 2X speedup with respect to LogTM-SE. The reasons why FASTM outperforms LogTM-SE in all the benchmarks are explained in the following paragraphs.

Fast abort recovery. FASTM decreases the time spent in abort recovery, which reduces overall execution time. As we can see in Figure 4.13, the LogTM-SE recovery mechanism accounts for 6.6% of the total execution time on high-contention applications and for 1.4% on low-contention applications. Note that, in coarse-grained applications like *Btree-var*, *List-long* or *Labyrinth*, up to 10% of the time is spent in the software abort routine. This undesirable overhead can be reduced if we apply a fast abort recovery mechanism. In fact, FASTM only spends, on average, 1.9% of the execution time to restore the pre-transactional state of high-contention applications, which corresponds to a 4.5X improvement over LogTM-SE. For low-contention applications, the time spent in abort recovery is negligible in FASTM.

Low conflict rate. By reducing the abort recovery time, FASTM decreases the number of conflicts that involve transactions that are aborting. In LogTM-SE, the transaction is alive until the very end of the abort recovery procedure. Thus, as we have shown in Figure 4.2, remote transactions that want to access to data owned by the aborting transaction will generate conflicts. As FASTM aborts transactions faster, most of the conflicts produced in the LogTM-SE abort period disappear. This benefit can be seen from the data in Table 4.1, which shows, for both HTM systems, the rate of aborts per transaction (labeled Abort Rate) in variable- and coarse-grained applications.

Figure 4.14 shows the speedup achieved by the FASTM-SIG (labeled S), the FASTM (labeled F) and the Ideal VM (labeled I) implementations over 32-threaded (low-contention applications, top of Figure 4.14) and to 16-threaded (high-contention applications, bottom of Figure 4.14) LogTM-SE executions. The average time in both graphics is calculated using the harmonic mean.

Small Write Signature. The figure shows that FASTM can also take advantage of the `T` state to reduce the pressure on signatures, which may lead to less false conflicts. However, this fact is not critical in the majority of the benchmarks. As can be seen in Figure 4.14, benchmarks with small or medium size transactions do not suffer from false positives when 2 Kbit signatures are used, and thus FASTM-SIG and FASTM obtain similar performance results. Only *Btree-var* and *Labyrinth*, which execute huge transactions, gain from this enhancement, showing a up to a 10% speedup in the comparison between FASTM-SIG and FASTM.

On the other hand, FASTM-SIG facilitates the use of mechanisms like those of LogTM-VSE [119] to survive context switches or page faults (because the write set of the transaction is already in the Write Signature). With FASTM, the Write Signature has to be reconstructed from the log (the hardware Write Signature does not include the `T` state lines in the L1 cache). Given that our evaluation shows that the fidelity of the Write Signature is not critical, FASTM-SIG may be a good alternative to simplify transaction virtualization.

Good fine- and coarse-grain performance. As it can be seen in the previous figures, fine-grained applications—those that execute small transactions—exhibit good scalability in the majority of TM systems given that most of their time is spent in non-transactional code. *Ssca2* does not show this behavior because most threads wait in barriers on certain phases of

	LogTM-SE			FASTM			FASTM-SL		
Bench	Commits	Abort Rate	SW Ab	OV Tx	Abort Rate	SW Ab	OV Tx	Aborts	SW Ab
Bayes	520	3.9	100%	71	3.4	16%	37	2.3	10%
Btree-var	2048	1.03	100%	627	0.22	6.5%	34	0.18	2.2%
Genome	19330	0.13	100%	318	0.11	0.6%	18	0.11	0%
Hash-read	4096	0.09	100%	1433	0.07	31%	334	0.07	0%
Intruder	22516	4.84	100%	450	3.36	0%	0	3.34	0%
Lists-long	8192	0.92	100%	802	0.53	7.7%	9	0.42	0%
Yada	2966	2.3	100%	511	2.01	2%	24	2.01	0.6%
Btree-fix	16384	0.3	100%	245	0.05	0%	1	0.03	0%
Hash-write	4096	0.77	100%	2123	0.56	44%	1970	0.53	10.6%
Labyrinth	4128	2.95	100%	2318	0.22	39%	379	0.22	1%
Vacation-low	16384	0.01	100%	820	0.01	0%	0	0.01	0%
Vacation-high	4096	0.38	100%	492	0.27	20%	272	0.21	4.5%

Table 4.1: Overflow, abort and software abort rates for variable- and coarse-grained 16-threaded executions under LogTM-SE, FASTM and FASTM-SL

the execution. In fact, LogTM-SE does not lose much performance in fine-grained applications due to their parallel nature. However, benchmarks with some contention, like *List-short* or *Raytrace*, are far from the upper-bound because more than 40% of the execution time is devoted to conflict management. In those applications, FASTM achieves similar performance to the Ideal VM implementation. This is because fine-grained applications almost never evict transactional cache lines, so no software aborts are performed.

Some coarse-grained applications, like *Genome*, *Hash-read* or *Vacation-low* scale well because they present few aborts. However, other applications with large transactions do not scale because most of the transactions conflict or overflow. For instance, applications that have lots of aborts, like *Hash-write*, *Intruder* or *Yada*, require a large number of backoff or stall cycles (up to 70%) in LogTM-SE.

In these benchmarks, the fast abort recovery of FASTM reduces the time wasted in non-useful transactional work, the time spent in stalled transactions and the time that processors execute the backoff (see Figure 4.13). Although most coarse-grained benchmarks, like *Bayes*, *Vacation-high* or *Yada*, have an important number of overflows, FASTM recovers the majority

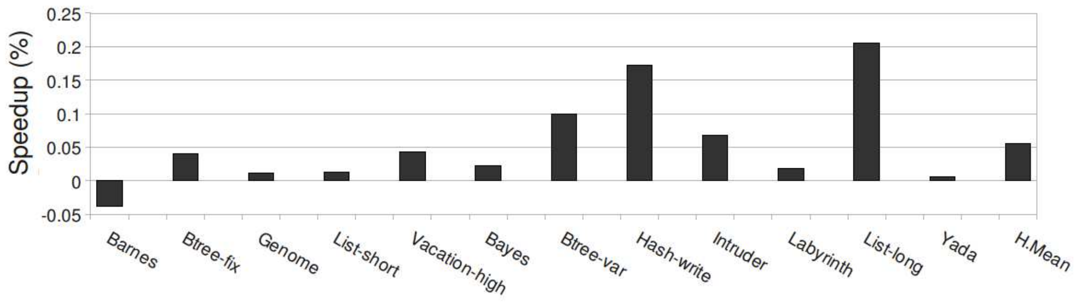


Figure 4.15: Speedup of FASTM-WN over FASTM in 16-threaded medium- and high-contention TM applications

of the aborted transactions almost immediately by hardware. This can be seen from the data in Table 4.1, where we can see the number of committed transactions (labeled Commits) and the number of transactions that evict transactional lines from the L1 cache (labeled OV Tx). We can also see in that table the percentage of aborts that are restored by software (labeled SW Ab).

As a result, FASTM performs comparable to the Ideal VM implementation for most of the applications (only 3% worse for low-contention benchmarks, 18% worse for high-contention benchmarks). However, some applications with large transactions and contention still execute slow aborts. For example, *Hash-write* and *Labyrinth* suffer a significant amount of software aborts—up to a 10% of the time is spent in abort recovery. As we will see in Section 4.7.3, selective logging is an attractive solution to improve on FASTM performance.

4.7.2 FASTM-WN Performance Analysis

For the performance analysis of FASTM-WN, we have focused on medium- and high-contention benchmarks because they typically report a gross amount of memory violations, so they are more sensitive to the impact of continuously retrying conflicting memory accesses or having extra network messages. We do not show the results of applications with a low conflict rate because most of their memory requests are not subjected to long delays produced by collisions among transactions.

Figure 4.15 presents the speedup of FASTM-WN over conventional FASTM . Both HTM systems have been evaluated with 16-threaded executions of medium- and high-contention benchmarks. As it can be seen in Figure 4.15, FASTM-WN outperforms all FASTM executions

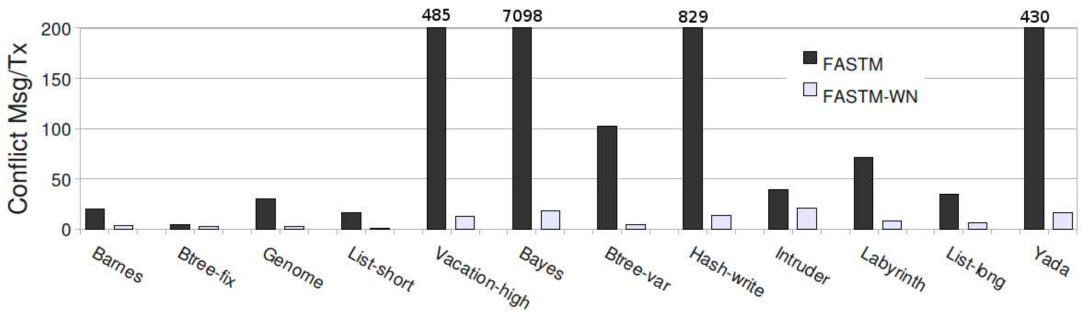


Figure 4.16: Network conflicting messages per transaction of 16-threaded medium- and high-contention TM applications in FASTM and FASTM-WN

except in *Barnes*, obtaining an average speedup of 5.5%. The benefit of wake-up notification is more notable in applications with large transactions like *Hash-write* or *List-long*, where the FASTM-WN system can devote all the on-chip resources to non-stalled transactions.

Remember that regular FASTM keeps retrying the conflicting access until the memory operation succeeds. Until then, the conflicting requests may block shared resources (*e.g.*, routers or directory entries) and thus increase the latency of conflict-free memory operations. In FASTM-WN, however, stalled transactions must wait until they receive the *WakeUp* message to retry the memory access. This procedure is counter-productive in fine-grained applications like *Barnes* because conflicting accesses must wait until they are notified to resume their execution. The reasons for such improvements are described below.

Reduction of network traffic. Performing wake-up notification on commits and aborts removes unnecessary network messages introduced by non-satisfied memory requests. Figure 4.16 shows the average number of messages (*Nack*, *WakeUp* and *Ack* in FASTM-WN, only *Nack* in FASTM) generated by conflicting accesses. Wake-up notification drastically reduces the amount of messages by a factor of 50X (in *Bayes* more than 500X). This fact accelerates still working transactions, which can commit (and eliminate the conflict) earlier.

Saving energy on conflicts. Disabling conflicting cores permits the system to reduce the number of "active" threads that keep interacting with shared resources. Figure 4.17 breaks down the system activity during the parallel execution showing the number of cores that are operative at a time. At it can be seen, in applications like *Hash-write* or *List-long* half or more processors

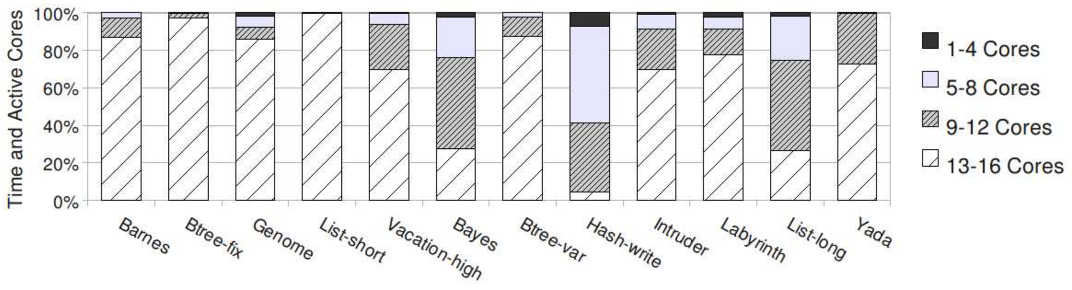


Figure 4.17: Number of active cores during 16-threaded medium- and high-contention executions in FASTM-WN

are stopped[4] during, at least, a 25% of the execution time. This strategy saves global power because disabling memory traffic on conflicts lowers the energy consumption. A power-hungry possible optimization is to spend this power budget for increasing the frequency of active cores in order to accelerate the execution of conflict-free transactions.

4.7.3 FASTM-SL Performance Analysis

For the performance analysis of FASTM-SL, we have selected variable- and coarse-grained benchmarks because they typically execute large transactions that overflow the L1 cache, and thus they are more sensitive to the VM strategy implemented in the base HTM system. We have omitted the results of applications with small (non-overflowing) transactions because they only report speedups between 1% to 3%, although they never perform worse in FASTM-SL. Table 4.1 provides detailed information about the applications we utilize and how they perform under LogTM-SE, FASTM and FASTM-SL. These numbers were collected running the applications with 16 threads.

Figure 4.18 presents the time distribution of FASTM (labeled F), FASTM-SL (labeled S) and Ideal (labeled I) HTM systems in their 16-threaded executions. The execution time has been normalized to the 16-threaded FASTM execution and is broken down using the same parameters than Figure 4.13. As it can be seen in Figure 4.18, FASTM-SL obtains a 18% speedup over FASTM (15% reduction of execution time), obtaining close performance to the Ideal VM ap-

[4]those cores are not completely disabled because registers, caches and other HTM support must preserve data.

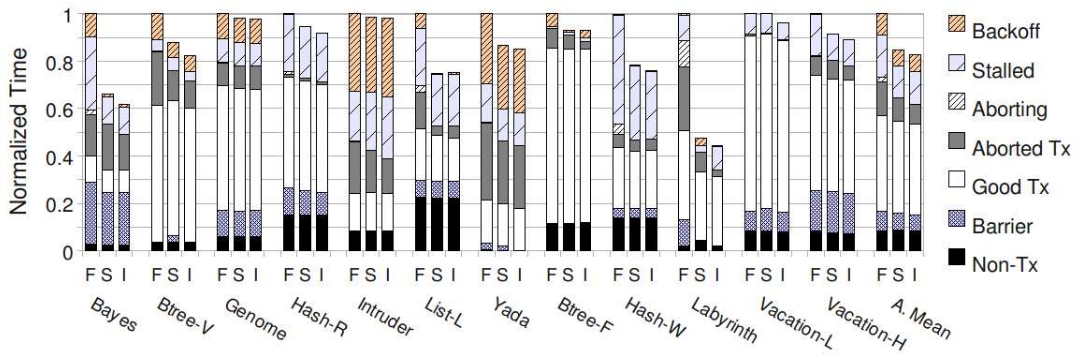


Figure 4.18: Normalized execution time of variable- and coarse-grained TM applications under 16-threaded FASTM (F), FASTM-SL (S), and Ideal (I) HTM systems

proach for all the benchmarks. The benefit is especially noticeable in *Bayes* or *Labyrinth*, which achieve almost 2X speedup over FASTM. The reasons for this behavior are the following.

Small log size. Selective logging drastically reduces the number of cache lines that have to be maintained in software. Figure 4.19 shows the average size (in KB) of the software log per transaction in FASTM and in FASTM-SL. Selective logging drastically lowers the size of the log by a factor of 15X (in *Hash-write* almost a 100X). This fact has two implications. First, there are less transactions that overflow the L1 cache (Table 4.1, labeled OV Tx). Second, as there is more space in the L1 for caching transactional data, the hit rate of the L1 cache increases higher.

Efficient transactional stores. In FASTM-SL, transactional stores do not need to access the software log each time they are retired—only when they leave the L1 cache, which is an uncommon event. As a result, the time spent in transactions that commit (Good Tx in Figure 4.18) is reduced by 3% on average.

Negligible software abort recovery. In case of abort, the software has to restore just a few lines. Moreover, the number of software aborts is also reduced in FASTM-SL because less transactions overflow the L1 cache (Table 4.1, labeled SW Ab). Accordingly, Figure 4.18 shows that FASTM-SL virtually eliminates the abort recovery overhead. As pointed out in the FASTM performance analysis, speeding up aborts cuts down the time that transactions are exposed to conflicts, which turns out to lower the abort rate (Table 4.1, labeled Abort Rate) and the time spent in Stall and Backoff cycles.

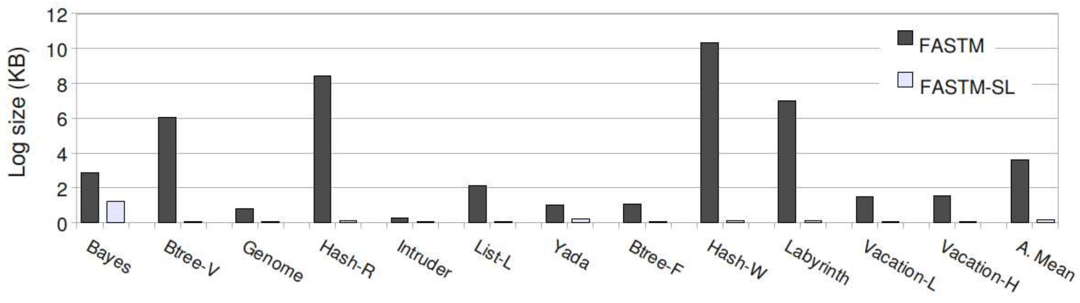


Figure 4.19: Software log size in FASTM and FASTM-SL

4.7.4 FASTM Conflict Resolution Analysis

The Stall conflict resolution policy sometimes exhibits pathological behavior that can affect the performance of the application [14]. For this reason, we have evaluated both LogTM-SE and FASTM with three other conflict resolution policies:

Abort: Aggressive policy that tries to eliminate the conflicts generated by stalled transactions. When a conflict is detected, the system aborts the requester, instead of stalling the transaction [98]. It also requires a backoff to avoid multiple aborts of transactions.

Timestamp: Policy that eliminates the backoff cycles by guaranteeing the progress of the oldest transaction, based on [98]. If a processor receives a conflicting request, it checks the remote timestamp and, if it is older than the local timestamp, the processor aborts the local transaction after sending a *Nack* to the requester together with its timestamp. When a processor receives a *Nack* message, it checks the remote timestamp and, if it is older than the local, it aborts the local transaction. Otherwise, it keeps issuing the request until the conflicting transaction finishes its abort recovery process.

Hybrid: Enhanced policy described as EE_{HP} in [14]. It works like the Stall policy, but write requests abort younger readers in order to directly get the ownership of the requested data. This policy eliminates the *starvation of the writer* pathology.

We have evaluated LogTM-SE with all the conflict resolution policies and we have found that the Stall policy outperforms the Abort and the Timestamp policy in LogTM-SE because it reduces the number of software aborts. However, LogTM-SE with the Hybrid policy achieves better results than LogTM-SE with the Stall policy in benchmarks with small transactions and

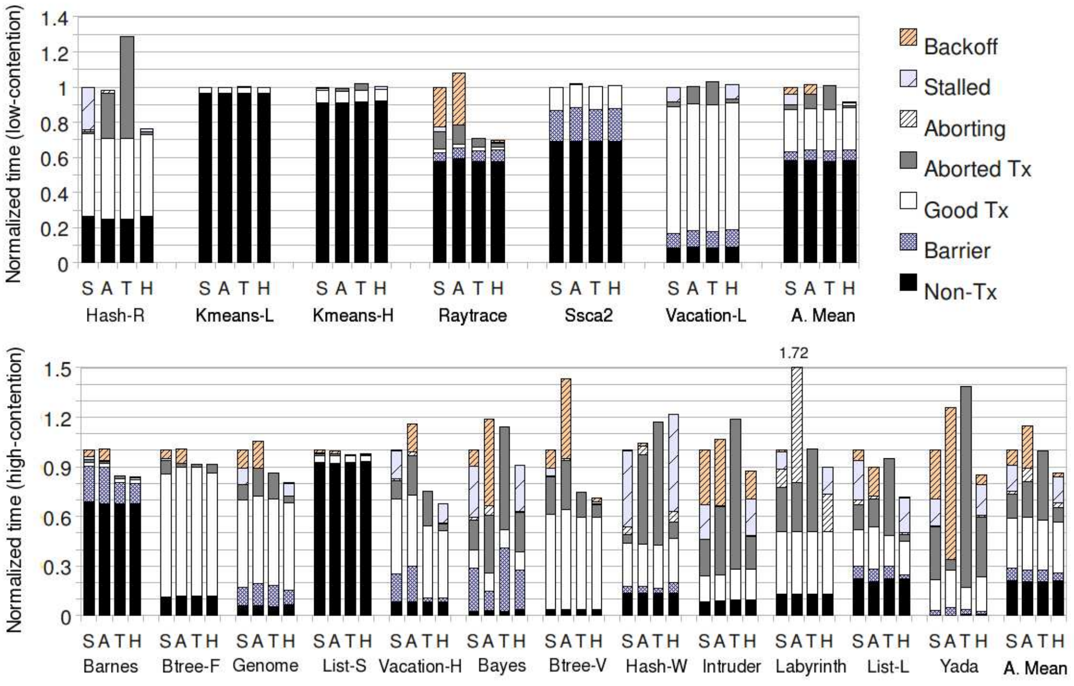


Figure 4.20: Distributed executed time of low-contention (top, 32 threads) and medium- and high-contention (bottom, 16 threads) TM applications under LogTM-SE (L), FASTM (F) and Ideal (I) HTM systems

high-contention, like *Barnes* or *List-short*, or in applications with read-only transactions, like *Hash-read* or *Btree-var*. In these situations, LogTM-SE with Hybrid obtains similar performance to FASTM given that most aborted transactions do not need to restore too many lines. However, LogTM-SE with the Stall policy presents better performance in applications with large transactions, like *Vacation-high* or *Yada*.

FASTM can take advantage of aggressive conflict resolution policies because it minimizes the impact of aborts. Figure 4.20 shows the time distribution of FASTM with Stall (labeled S), Abort (labeled A), Timestamp (labeled T) and Hybrid (labeled H) conflict resolution policies normalized to the 32-threaded (low-contention) and 16-threaded (high-contention) execution of FASTM with the Stall conflict resolution policy.

The Abort policy removes stalling transactions in case of conflict given that transactions automatically abort. *List-long* can benefit from this policy, because conflicts that involve stalled transactions disappear. However, in benchmarks with high-contention and small transactions, like *Raytrace* or *Genome*, the number of aborts augments significantly, increasing the time spent in backoff. Moreover, it also increases the number of aborts that have to be recovered by software, what is critical in applications like *Labyrinth*

The Timestamp policy improves some high-contention benchmarks with variable-size transactions, like *Btree-fix* or *Vacation-high*, because it does not require backoff cycles. Nonetheless, the Timestamp policy has some weaknesses. First, it constantly aborts transactions, which increases considerably the discarded work in coarse-grained applications like *Bayes*. Second, a transaction remains stalled until the younger conflicting transaction finishes its abort phase. Although FASTM provides fast abort recovery, those transactions that overflow the L1 cache do not abort instantaneously. This is a problem in benchmarks that continuously execute large transactions (*e.g.*., *Yada*), given that overflowing transactions must abort each time they find a conflict using the slow software routine.

The Hybrid policy improves our baseline because it reduces the starvation of older writers without increasing contention. Like in LogTM-SE, the Hybrid policy accelerates applications with high-contention and small/read-only transactions. Moreover, the fast abort recovery mechanism allows FASTM to improve the performance of some coarse-grained benchmarks as well, like *Intruder* or *Yada*, which discard a lot of work when large transactions abort.

4.8 Related Work on Eager HTM Systems

Unbounded TM (UTM [5]) was the first HTM that allowed a fast (non-software) execution for transactions of any size or duration. UTM extends each memory block with R/W bits and an address pointing to an entry of the a hardware-accessed XSTATE structure. Because UTM implements *early* version management, speculative updated values are coupled with a pointer that indicates where the original data is—this data has to be restored in case of abort. Thus, XSTATE contains a linked list of transactional accessed (either read or written) addresses and a status register of the current transaction. Those linked lists must be traversed each time a memory operation finds the R/W bit set (to identify which transaction owns a block) and at commit or abort time (to update the global state). Although UTM does not require transactional caching support for correctness, it can be used for speeding up speculative execution.

Large TM (LTM [5]) presents a simplification of the UTM engine. In LTM, a per-set *overflow* bit in the L1 cache informs ongoing memory operations that an in-flight transaction has evicted a line. Thus, overflow storage has to be accessed when memory operations find the overflow bit marked. In these occasions, the processor interrupts the execution and triggers a walk in a hash table kept in DRAM memory. This happens when loads miss their local cache (as they need to read the value generated *earlier* in the transaction) or when remote requests have a potential conflict with the line (as they must be denied because LTM resolves conflicts *eagerly*). On commits, LTM writes back overflowed data from the overflow storage space to the main memory.

Virtualized Transactional Memory (VTM [97]) tracks transactional information—*e.g.*, speculative values or read and write sets—in a table placed in application's virtual and private memory, called XADT. VTM uses conventional (bounded) HTM caching support for small transactions, invoking virtual machinery only when it is necessary. Like in LTM, the XADT is accessed on L1 misses (either local or remote) and at commit time, but, instead of relying on per-block metadata pointers, VTM extends processors with microcode (or firmware) aptitudes to operate in software structures. Performing lookups on each L1 miss incurs in a significant slow down. To overcome this issue, VTM introduces a software-managed Bloom filter, which conservatively dictates if a given address is present in the XADT. Additionally, VTM can incorporate a transactional victim cache to keep constantly-accessed data close to the processor. Hence, hardware requirements of VTM are not as expensive as UTM, while they provide transparent execution of unbounded transactions and capability for handling transactions in the presence of context switches and page faults.

Page-based Transactional Memory (PTM [24]) expands an LTM-like HTM system with shadow pages that hold transactionally modified values. An additional table maps physical pages to shadow pages, while processors maintain in the specialized vectors which page entries have been accessed within a transaction, a requirement to perform conflict detection. Shadows pages must be copied in their associated physical pages at commit time to make the state globally visible.

LogTM [84] simplified the UTM mechanism by storing old values and their associated address in a private log. Like other HTM systems, it uses L1 R/W bits to track those memory

HTM System	VM Strategy	Hardware Support	Abort Recovery	Overflow Policy	Commit Process	CM Strategy
LogTM-SE [130]	Early	Logging	Software	Update Memory	-	Eager
Rock HTM [33]	Late	Store Buffer	Hardware	Notify Software	Drain Buffer	Eager
HyTMs [31, 63]	Late	L1 TX Cache	Hardware	Run STM	Update Memory	Eager
VTM [97]	Late	L1 TX Cache	Hardware	Software Structure	Update Memory	Eager
FASTM	Early[5]	Logging, L1 TX Cache	Hardware and Software	Update Memory	Clean L1 State	Eager

Table 4.2: Data VM characteristics of eager HTM systems

accesses performed inside a transaction, albeit it modifies the coherence protocol with sticky directory states to perpetuate the partnership between a transactionally evicted line and its last owner. Sticky directory states forward memory request to the last owner(s) of the line, which tests its R/W bits to perform conflict detection—even in the case they have evicted the cache line—and reply with a *Nack* message in case of conflict, instead of acknowledging or sending the data to the requester. The W bit is also used to prevent repetitive logging when a memory block is updated multiple times. Notice that, in contrast to UTM, LogTM does not require per-block metadata for data versioning purposes.

LogTM-SE [130] decouples transactional state from caches, replacing the L1 R/W bits of LogTM with signatures that summarize those *physical* addresses accessed within a transaction. This implementation also uses sticky states for those data evicted in the L1 cache but present in the L2 cache. However, L2 evicted data is forwarded to all the processors of the CMP, which test their signatures and reply with a *Nack* message if a conflict occurs. LogTM-SE cannot use the W bit to determine if a given address has been introduced in the log. Instead, it uses an effective small table containing recent logged addresses. LogTM-VSE [119] shows how

[5]we consider FASTM an early VM system, although it shares some similarities with late VM system

signatures can be in *virtually* summarized using OS support to keep executing transactions after they are de-scheduled or interrupted by a page fault.

In [69], we revealed how log-based HTM systems can be accelerated by using an in-core gated store buffer, not unlike how the implementation proposed for Rock [33] keeps the transactional state. This approach is extremely encouraging because it allows flexible *early* and *late* version management. Moreover, the industry has voted for introducing exposed write buffers in the microarchitecture of future CMPs, which makes this approach even more feasible.

Table 4.2 summarizes the main VM characteristics of state-of-the-art eager HTM systems and compares them to FASTM. As it can be seen, FASTM presents fast commit and abort procedures with reasonable hardware support and minimum complexity, a feature that is missing in other unbounded eager HTM systems.

4.9 Conclusions

In this chapter, we have presented FASTM, the first log-based HTM system that, like late VM approaches, takes advantage of the processor's cache hierarchy to provide fast abort recovery. FASTM uses a novel coherence protocol to buffer the transactional modifications in the first level cache and to keep the non-speculative values in the higher levels of the memory hierarchy. This mechanism accelerates the abort recovery of large transactions, which is critical in other log-based implementations like LogTM-SE.

To handle cache overflows, FASTM follows a log-based approach. Transactional cache lines are evicted in-place in the memory hierarchy and old values are maintained in a cacheable log, which must be restored by a software routine. This approach simplifies overflow mechanisms of late VM systems, that either need complex specialized hardware to handle cache misses and to commit overflowed lines or fall-back to software-only transactions.

We have evaluated FASTM with a heterogeneous set of applications and conflict resolution policies. Our proposal obtains, on average, a speedup of 44% over LogTM-SE in high-contention applications. We have seen that the performance improvement is more pronounced in applications with coarse-grain transactions, because FASTM reduces considerably the time spent in abort recovery as well as the number of conflicts. Although our analysis shows that transactional cache replacements are common in coarse-grain applications, FASTM does not suffer performance penalties, because transactions that overflow the caches do not usually abort.

We have also proposed two additional optimizations for FASTM: wake-up notification and selective logging. While the former proposes an efficient solution to handle stalls in eager HTM systems, the latter reduces the pressure on logging mechanism in FASTM. Our evaluation shows that selective logging accelerates transactional execution, reduces the number of slow aborts and decrements the size of the software log, achieving an average speedup of 18% over FASTM. On the other hand, the wake-up mechanism delivers good performance (5.5% speedup over FASTM) and saves energy in the system when threads are stalled by large transactions.

Our evaluation of FASTM with different conflict resolution policies shows that having a fast abort recovery mechanism favors aggressive policies that abort critical transactions in situations with high-contention.

Chapter 5

Speculative Hardware Transactional Memory Systems with Local Commits

Conflict management (CM) is possibly the most critical feature of HTM systems [38]. The literature is full of proposals that try to improve on this mechanism, either by moving the resolution of conflicts to software [103, 112], applying high-performance hardware policies [14, 98, 111] or modifying the coherence protocol drastically [7, 99, 124].

Lazy HTM[1] systems [22, 45, 92, 107, 112, 124] commonly obtain better performance than eager HTM systems [14, 111, 122] because they (i) offer more concurrency (transactions speculate with conflicts), (ii) guarantee forward progress (no backoff or time-based policies are required) and (iii) permit transactional readers to overlap their execution with non-committed writes, which removes direct conflicts if a transaction that reads a memory location finishes before a transaction that writes the same location.

Unfortunately, prior lazy HTM systems suffer numerous limitations that may affect both the scalability and the complexity of the system. In lazy HTM systems, transactions require arbitration and data movements at commit time, which incurs significant overheads. They also introduce sophisticated commit protocols that are quite hardware invasive—they demand several changes in the communication between private and shared resources. Moreover, lazy HTM systems impose *late* VM, and thus they are subjected to long delays when they execute transactions that commonly exceed on-chip buffers, as explained in Chapter 4.

[1]in this thesis, we refer as lazy HTMs those systems that speculate with conflicts and resolve conflicts at commit time, independently of how conflicts are detected or the VM strategy that they employ.

In order to address the above issues, this chapter starts presenting FuseTM, a fused dual-mode HTM system with local commits. This is a speculative HTM system that integrates *lazy* resolution of conflicts in a conventional *eager* HTM framework. By adding minor changes in the coherence engine, FuseTM keeps executing after a conflict occurs and performs *local* commits—a technique that **moves arbitration, data transfers and directory updates out of the critical path**. Moreover, FuseTM is the first speculative approach that offers **simultaneous execution of eager and lazy transactions** in the same latent microarchitecture. Hence, FuseTM breaks with the *late* VM invariant for lazy HTM systems: the system switches to the eager (log-based) execution mode when a transaction exceeds the L1 cache in order to maintain in-place in memory the overflowing state. This strategy simplifies the hardware design that is required to support unbounded transactions.

FuseTM forbids lazy execution for those transactions that evict speculative data from the L1 cache—large transactions are aborted and restart in eager mode. This can be an issue if overflows are frequent, because the system must discard lot of useful work and enforce eager CM from the very beggining, which results into less concurrency among conflicting transactions. To overcome the above problems, we implement SpecTM, a speculative HTM system with early overflowing updates. SpecTM shares most of the underlying infrastructure of FuseTM—and thus most of its features—but it has one major difference. By readjusting the selective logging mechanism, SpecTM is able to **keep running in lazy-mode until commit time**. This technique removes aborts provoked by overflowing transactions and performing enables resolution of conflicts for most of the transactionally accessed lines.

This chapter starts presenting the motivation for FuseTM and SpecTM, showing the weaknesses of contemporary lazy HTM systems. Then, it overviews the FuseTM system, describing the hardware extensions and the unified coherence protocol that FuseTM (and SpecTM) utilizes. The chapter follows up summarizing how FuseTM operates in the lazy mode, and then presents how the system is able to combine eager and lazy transactions without requiring cumbersome hardware. Then, the chapter goes on overviewing SpecTM, explaining the additional mechanisms that it requires and how they operate. The chapter finalizes evaluating FuseTM and SpecTM, and comparing both systems with other modern commit protocols for lazy HTM systems.

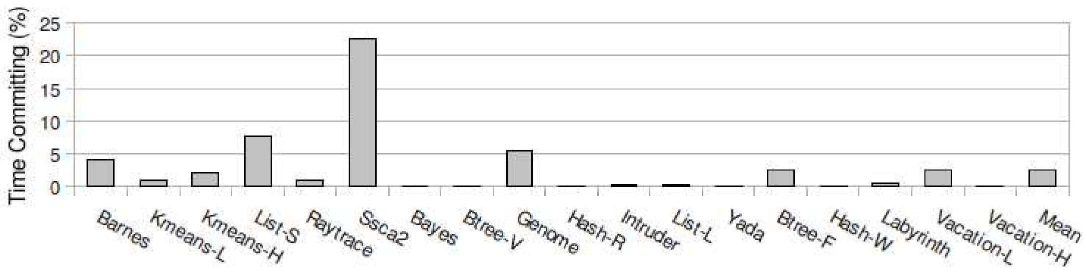


Figure 5.1: Percentage of time spent in arbitration under 32-threaded TCC-Dist

5.1 Motivation

Several studies [14, 111] showed the benefit of using lazy management of conflicts when dealing with high-contention transactions. However, committing transactions is an expensive operation when the resolution of conflicts is moved at the end of a transaction because it requires arbitration. Software arbitration [112] produces important delays at commit time. On the other hand, hardware arbitration [21, 22, 45, 92] serializes transactional computation, which compromises the scalability of the system.

In order to quantify the performance loss of a standard commit protocol, we use a TCC-based implementation with distributed commits as a lazy HTM baseline (TCC-Dist for short). This is a lazy HTM system that assumes an idealized late VM mechanism (zero-cost commits and aborts) and an instantaneous abort notification (inconsistent transactions undo their modifications at the same time that a transaction commits). TCC-Dist uses the distributed algorithm presented in [92] to arbitrate between committing transactions. More documentation regarding the lazy HTM baseline can be found in Section 3.2.2.

Fine-grained transactional workloads suffer important overheads when arbitration is required. Figure 5.1 shows the percentage of time spent in arbitration after executing 32-threaded TM applications using our reference HTM system. Applications are gathered according to the size of their transactions; from fine-grained (left) to coarse-grained (right). As it can be seen, TM applications waste substantial part of their time in commit arbitration, especially those that continuously execute tiny transactions (*e.g.*, *Genome*, *List-short* or *Ssca2*).

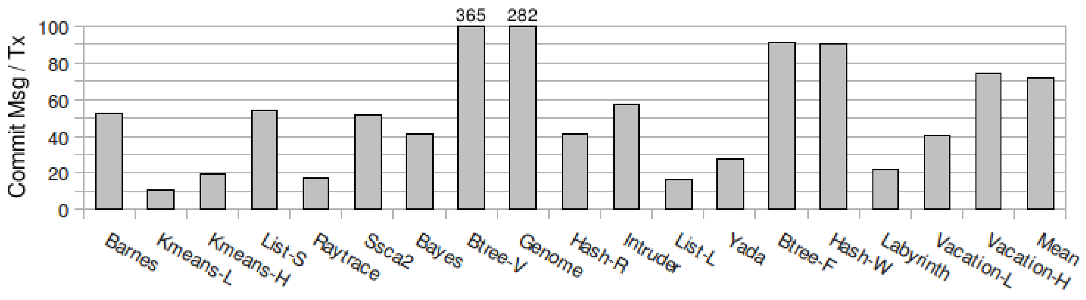


Figure 5.2: Average network messages in the commit phase under 32-threaded TCC-Dist

The reason of such commit delay resides on the communication with shared resources—in TCC-Dist, transactions acquire/release directory modules. Figure 5.2 shows the average number of messages per transaction introduced in the network at commit time. As it can be seen, variable- and coarse-grained applications introduce more messages because they read and write more directory banks. Nonetheless, the commit phase does not become critical because most of the time is devoted to compute large transactions. However, applications with small read/write sets may require numerous messages to acquire a single directory module, given that in most of the occasions the module is already acquired (the *Acquire* message must be re-issued).

Commit arbitration is a not the only problem that lazy HTM systems have to face. For instance, TCC implementations [22, 45] require write backs to the shared L2 cache, while EazyHTM [124] needs atomic directory updates. As a matter of fact, all the proposed approaches introduce substantial changes in the CMP configuration. This includes a firmware that walks the entries of the L1 cache to identify those that have been transactionally written, a mechanism that sends bulk messages containing the write set to remote cores, the L2 cache and/or the directory, a deadlock-free protocol that groups and blocks directory modules and a device to support atomic memory updates. Hence, lazy HTM systems present *ad hoc* implementations that highly depend on the underlying hardware machinery.

Finally, lazy HTM systems assume late data versioning for overflowed data—they keep overflowed data hidden from in-flight transactions using specialized structures, such as firmware-accessed memory structures (LTM [5] or VTM [97]), shadow memory pages (PTM [24] or XTM [25]) or additional hardware tables (FlexTM [112] or EazyHTM [124]). When the transactional buffers are overflowed, the system inserts new data in these specialized structures, where it is kept until the transaction commits (new data is transferred to global memory) or aborts (new data is invalidated). Also, if the transaction does not find the data in the transactional buffers, lazy HTM systems must traverse specialized structures to check if the accessed data has been modified during the in-flight transaction. In contrast to early VM systems, late VM systems are subjected to long delays when they execute transactions that commonly exceed on-chip data versioning support.

The previous paragraphs reflect three major concerns that affect the performance and the design of lazy HTM systems. First, arbitration is an expensive operation for fine-grained transactional applications; therefore it would be extremely helpful to eliminate it from the commit phase. Second, the design of a lazy HTM should not rely on specific hardware components, nor include atomic operations at commit time. Third, using late data versioning for holding the overflowing state adds even more obstacles in an already complex design; a simpler approach is worthy to be preferred.

5.2 A Fused HTM System with Local Commits

FUSETM presents a novel HTM implementation to overcome the above issues. This proposal slightly modifies the L1 cache controller to simultaneously combine eager and lazy transactions and provide *local* commits, which avoids expensive commit arbitration [92] and directory updates [124], while it keeps standard activity for regular memory operations.

5.2.1 FUSETM Overview

FUSETM offers two different execution modes: eager and lazy. The *eager* FUSETM execution mode uses both eager version and conflict management. The *lazy* FUSETM execution mode, on the other hand, uses both lazy version and conflict management. Moreover, FUSETM permits eager- and lazy-mode transactions to execute *simultaneously* in the system. This is possible through UTCP, a novel unified transactional cache coherence protocol that is able to correctly track conflicts among transactions—independent of their execution mode—and it ensures the correct propagation of transactional modifications.

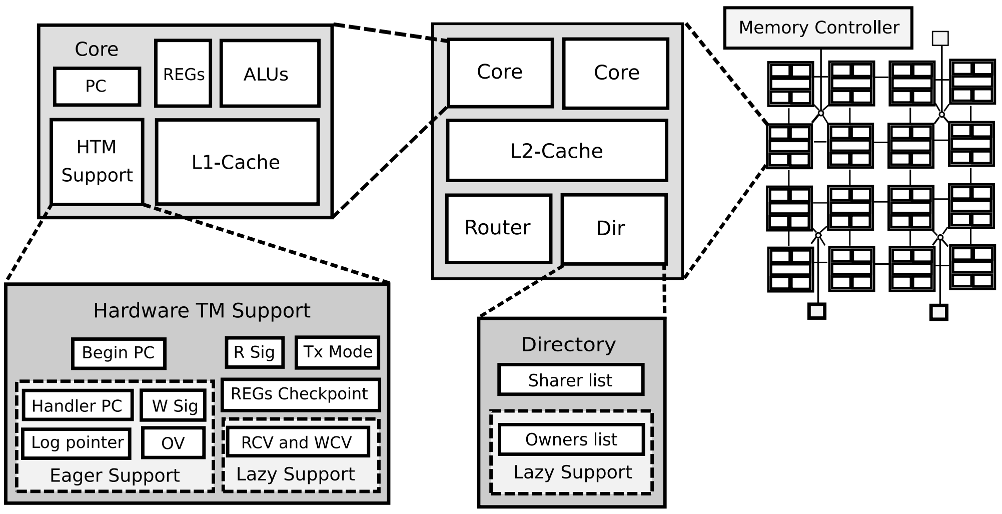


Figure 5.3: Base system configuration and transactional hardware support for DYNTM

The FUSETM eager execution follows a log-based approach. Transactional modifications are kept in-place in memory, where they are allowed to propagate to all levels of the hierarchy. The pre-transactional state is logged in a software structure [84]. In the eager mode, conflicts are resolved as soon as they are produced. In contrast, the FUSETM lazy execution mode resolves conflicts at the very end of a transaction. In the lazy mode, the speculative state is buffered in the L1 cache and is not made visible to the rest of the system until the transaction is committed. FUSETM takes advantage of the built-in hardware support for eager version management in order to handle L1 cache overflows and context switches for lazy transactions. In such cases, the system will simply abort the lazy transaction and re-execute it in eager mode.

5.2.2 Hardware Support

In this work, we assume for our FUSETM implementation a CMP system similar to that utilized in the previous chapter (see Figure 5.3). Besides the UTCP coherence protocol, FUSETM requires additional extensions to existing hardware components for executing eager and lazy transactions:

Logging Support. Like previous log-based HTM proposals [84, 130], FUSETM extends the core with register checkpointing and *conventional* software logging support to implement early version management.

Signatures. FUSETM requires Read and Write signatures [21, 130] (Bloom filters) to track transactional accesses. While the Read Signature summarizes any transactional read, the Write Signature only contains addresses from eager transactional stores.

Conflict Vectors. Like FlexTM [112] or EazyHTM [124], DYNTM introduces Read Conflict Vector (RCV) and Write Conflict Vector (WCV) to maintain inconsistencies between in-flight transactions.

5.2.3 FUSETM Modes of Execution

Like other lazy HTM protocols, FUSETM restricts transactional updates to the L1 cache only, maintaining pre-transactional values in the L2 cache. However, rather than requiring specialized hardware to handle L1 cache overflows [97, 112], FUSETM aborts the offending transaction and re-executes it in eager mode. In this section, we overview both eager and lazy execution modes.

Eager execution mode. In FUSETM, eager transactions follow the same hybrid data version management mechanism as the one presented in FASTM. This mechanism guarantees that, if a transaction has not overflowed the L1 cache, the L2 cache will contain the correct pre-transactional state. This is done by writing back each *dirty* non-transactional L1 cache line before overwriting it with transactional data. By keeping both the old and the new (transactional) state in-place in memory, FUSETM offers a very fast abort recovery mechanism for transactions that do not overflow the L1 cache—it simply invalidates transactionally accessed lines. Eager transactions also maintain the old state in a private, cacheable software log [84], which permits the safe eviction of consistent transactionally written lines. In case of overflow, the pre-transactional state can be recovered by a software routine (slow abort recovery mechanism). Moreover, transactional store operations always add their addresses in the Write Signature. Thus, the FUSETM eager mode allows transactions to survive context switches and page faults by virtualizing the signatures and by using the software log for abort recovery [130]. FUSETM detects conflicts *early* with the help of the UTCP protocol.

Lazy execution mode. Lazy transactions also detect conflicts *early* via the UTCP protocol. Contrary to the eager execution mode, lazy transactions continue executing after detecting a conflict—conflicts are resolved *lazily* at commit time or until someone aborts the transaction. In order to track conflicts from their detection until their resolution time, FUSETM transitions conflicting cache lines to special UTCP states, and marks conflicts among cores in the Read and Write Conflict Vectors (RCV and WCV). In FUSETM, conflicts are notified at commit time using point-to-point *AbortTx* messages. This can be done because information about inconsistent transactions are recorded in the WCV. After acknowledging that all conflicting transaction have been aborted, the core *locally* commits the transactional data in order to make it *globally* visible. Unlike prior proposals, FUSETM does not require directory updates [20, 22, 124] nor data movement [45, 92, 112] at commit time.

5.2.4 The Unified Transactional L1 Cache Coherence Protocol

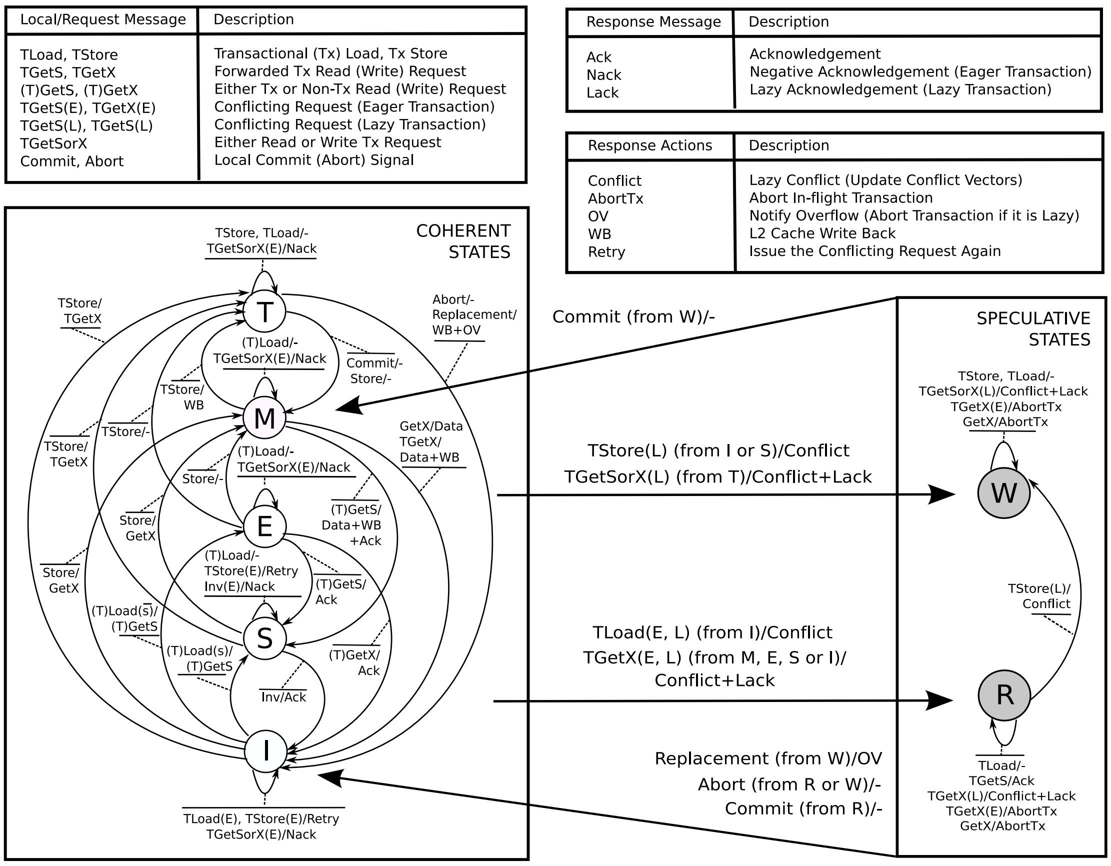
In the heart of FUSETM lies a novel coherency protocol, the Unified Transactional Coherence Protocol (UTCP), that guarantees the correct propagation of transactional modifications, as well as the prompt detection of conflicts among transactions.

The UTCP protocol distinguishes between coherent and speculative states. The coherent states include the four states of a typical MESI protocol, plus the T state. Cache lines in these states are either non-transactional or they are read inside a transaction and have no conflicts (M, E, S and I states), or they are written inside a transaction and they have no conflicts (T state). Notice that this is analogous to the Transactional Coherence Protocol implemented by FASTM.

The two speculative R and W states keep transactionally read (R) or written (W) cache lines that have a conflict with one or more other transactions. Cache lines are transitioned to the R or W states only when they have a conflict with a lazy transaction—eager transactions are not allowed to speculate with their execution when they conflict with other eager transactions.

The coherent states T, M, E have a single owner or version in the system directory (multiple sharers are allowed for S, of course). On the other hand, speculative lines can have multiple active versions, therefore the directory must maintain a vector of owners. Conflicts among transactions are detected through the and T and W states (for lazy transactional writers) and the Read and Write signatures (for transactional readers and eager transactional writers, respectively).

The UTCP protocol differentiates between eager and lazy memory requests by adding an extra bit in the coherence messages. Eager transactions notify conflicts using *Nack* messages. This mechanism allows eager transactions to maintain their isolation by preventing remote requesters to access their read/write sets. Requesting cores may retry the memory access or abort



Local/Request Message	Description
TLoad, TStore	Transactional (Tx) Load, Tx Store
TGetS, TGetX	Forwarded Tx Read (Write) Request
(T)GetS, (T)GetX	Either Tx or Non-Tx Read (Write) Request
TGetS(E), TGetX(E)	Conflicting Request (Eager Transaction)
TGetS(L), TGetS(L)	Conflicting Request (Lazy Transaction)
TGetSorX	Either Read or Write Tx Request
Commit, Abort	Local Commit (Abort) Signal

Response Message	Description
Ack	Acknowledgement
Nack	Negative Acknowledgement (Eager Transaction)
Lack	Lazy Acknowledgement (Lazy Transaction)

Response Actions	Description
Conflict	Lazy Conflict (Update Conflict Vectors)
AbortTx	Abort In-flight Transaction
OV	Notify Overflow (Abort Transaction if it is Lazy)
WB	L2 Cache Write Back
Retry	Issue the Conflicting Request Again

COHERENT STATES
TStore, TLoad/- TGetSorX(E)/Nack
T
M
E
S
I
TStore/ TGetX
(T)Load/- TGetSorX(E)/Nack
Commit/- Store/-
Abort/- Replacement/ WB+OV
TStore/ WB
TStore/-
TStore/ TGetX
GetX/Data TGetX/ Data+WB
(T)Load/- TGetSorX(E)/Nack
Store/-
(T)GetS/ Data+WB +Ack
Store/ GetX
(T)Load/- TStore(E)/Retry Inv(E)/Nack
(T)GetS/ Ack
Store/ GetX
(T)Load(s)/ (T)GetS
(T)Load(s)/ (T)GetS
(T)GetX/ Ack
Inv/Ack
TLoad(E), TStore(E)/Retry TGetSorX(E)/Nack
Commit (from W)/-
TStore(L) (from I or S)/Conflict
TGetSorX(L) (from T)/Conflict+Lack
TLoad(E, L) (from I)/Conflict
TGetX(E, L) (from M, E, S or I)/ Conflict+Lack
Replacement (from W)/OV
Abort (from R or W)/-
Commit (from R)/-
SPECULATIVE STATES
TStore, TLoad/- TGetSorX(L)/Conflict+Lack TGetX(E)/AbortTx GetX/AbortTx
W
TStore(L)/ Conflict
R
TLoad/- TGetS/Ack TGetX(L)/Conflict+Lack TGetX(E)/AbortTx GetX/AbortTx

Figure 5.4: State-transition diagram of the unified transactional L1 cache coherence protocol

the transaction. Nevertheless, when a core that executes a transaction receives a conflicting lazy request, it must transit the conflicting line to a speculative state and reply with a *Lack* message. In order to implement strong isolation, the UTCP protocol aborts transactions that receive non-transactional conflicting messages.

Figure 5.4 shows the UTCP states and transitions. The label of each transition shows the UTCP triggering message (before the slash) and the associated actions (after the slash). *TStore* and *TLoad* are memory accesses produced within a transaction. Note that a transactional L1 miss generates a memory request to the directory, which is forwarded to the owner(s) of the line (*TGetS* for reads, *TGetX* for writes). If the line has accessed the line within a lazy transaction—represented with the suffix (L) in Figure 5.4—the line transits to a speculative state and the receiver replies sending a *Lack* message to the requester. If the line has been accessed inside an eager transaction—represented with the suffix (E) in Figure 5.4—the receiver replies with a *Nack* message and the line remains in the same cache state.

The Conflict action updates the RCV (for R-state lines) or the WCV (for W-state lines) by marking the requesting/replying core in one of the conflict lists. The WB action pushes the line to the L2 cache. *Replacement* indicates an L1 cache eviction, which activates the abort machinery in lazy transactions (OV actions). Commit and Abort actions also trigger transitions to the Modified or Invalid state. A detailed explanation of lazy memory operations and how they affect the UTCP transitions is presented in the next section.

5.2.5 FuseTM Lazy Transactional Operations

This section explains how the lazy mode of FuseTM works. We describe how lazy-mode transactions interact with the elements of the system (UTCP protocol included). We omit here how eager-mode transactions operate, as they follow the same steps that in FasTM. We defer the explanation of how FuseTM mingle different-mode transactions on top of the same infrastructure to Section 5.2.7.

5.2.5.1 Lazy Transactional Stores

In FuseTM, non-conflicting transactional stores (*TStores*) follow the TMESI protocol proposed in FasTM—they write back the value of M-state lines to the L2 cache before transitioning to the T state. This guarantees that the L2 cache always has the correct pre-transactional value of the line.

The novelty of the UTCP protocol lies on how the system handles transactional conflicting lines. Let's assume that core L_0, which is executing a lazy transaction, attempts to write a line that has been accessed by other cores during their in-flight lazy transaction. In this scenario, L_0 requests the line to the directory, which (1) forwards the coherence request to the current owner(s) of the line and (2) sends a message containing the number of owners to L_0.

Now we assume that core L_1 is one of the *lazy* owners of that line. When L_1 receives the conflicting request (*TGetX*), it replies to L_0 with a *Lack* message and moves the line to one of the speculative states. If L_1 has transactionally written the line before (*i.e.*, the line is in the T or

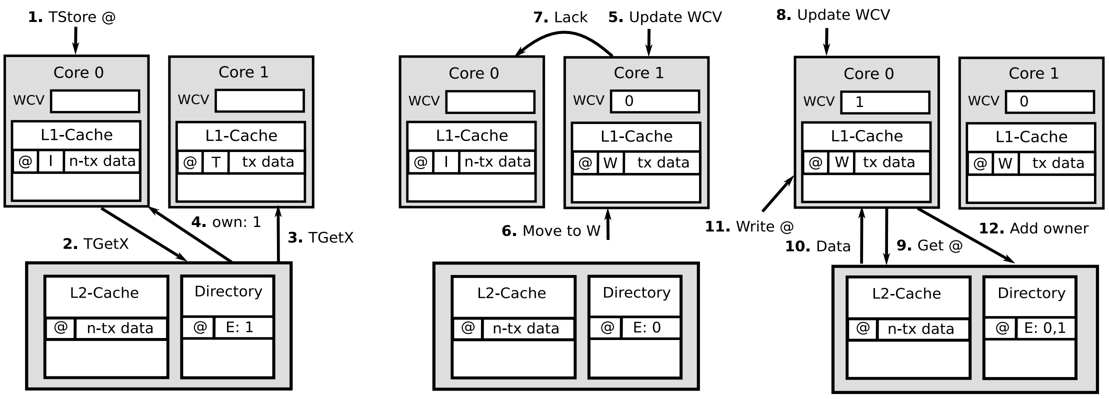


Figure 5.5: Conflicting transactional stores in FUSETM

W state), the line transitions to W and L_1 adds L_0 to its WCV. Similarly, if L_1 has transactionally read the line (*i.e.*, the line is in the M, E, S or R), the line transitions to R and L_1 adds L_0 to its RCV. When L_0 receives the *Lack* reply from L_1, it detects that there is a conflict. Hence, L_0's request is serviced by the L2 cache, which is guaranteed to have the correct pre-transactional values, and L_0 is added as a new line owner in the directory. L_0 puts the line in W and adds L_1 to its WCV. This mechanism permits the system to identify inconsistent transactions that should be aborted before committing.

In Figure 5.5 we can see an example where core L_0 misses in the cache after retiring a *TStore* (step 1). Core L_0 requests the line to the directory, which forwards the requests to the only owner of the line (in this case, core L_1 (step 2 and 3)). At that time, the directory notifies L_0 that a unique copy of the line is present in the system (step 4). L_1 has the line in the T state, so a conflict is detected. Hence, L_1 updates its WCV by marking the inconsistency with L_0 (step 5), transits the line to the speculative W state (step 6) and sends a *Lack* message to L_0 (step 7).

Once L_0 receives the *Lack* message, it adds L_1 in its WCV (step 8) and loads the line from the L2 cache, which has a consistent copy of the pre-transactional values of the line (step 9 and 10). After that, it performs the transactional write, moves the line to the W state (step 11) and unblocks the directory, setting itself as a new owner of the line (step 12).

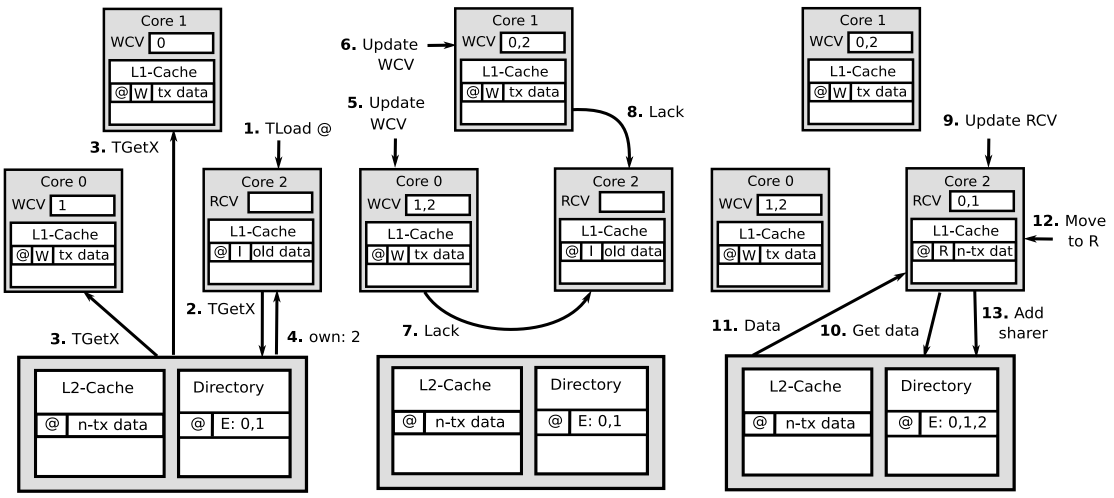


Figure 5.6: Conflicting transactional loads in FUSETM

5.2.5.2 Lazy Transactional Loads

Non-conflicting transactional loads (*TLoads*) are performed as regular loads (adding the address in the Read signature if they end successfully). However, conflicting transactional loads are executed following a similar strategy to *TStores*.

Let's assume that core L_0 attempts to read a line that has been written by another lazy transaction. If the line is in the R state, the pre-transactional value is already load in the L1 cache and the conflict has been detected as well, so the *TLoad* is completed. If the line is not valid in the L1 cache, L_0 requests the line to the directory, who forwards the requests to the owner(s) of the line. Then, all the writers of the line respond with a *Lack* message, add L_0 as a conflict in their WCV and transit to the W state—readers do not have to perform any action.

When L_0 receives the *Lack* replies, it marks the conflicts in its RCV and gets the old value of the line from the L2 cache, holding the line in the R state. Afterwards, L_0 communicates the directory that it owns another alive copy of the line.

Figure 5.6 shows an example of how FUSETM handles a transactional load that misses the L1 cache in core L_2 (step 1). L_2 requests the line to the directory, which transmits the request to L_0 and L_1, who have written the line before. The directory also sends a message to L_2 informing that the line has two owners (step 4). L_0 and L_1 own the line in the W state; therefore both cores mark L_2 in their WCV (step 5 and 6), and reply with a *Lack* message to L_2 (step 7 and 8). When

L_2 receives the two *Lack* messages, it marks L_0 and L_1 as potential conflicts in its RCV (step 9) and gets the old values of the line from the L2 cache, which is kept in the R state (step 10-12). Finally, it updates the directory, which contains now L_0, L_1 and L_2 as the current owners of the line (step 13).

5.2.5.3 Local Commits

In FUSETM, when a lazy transaction attempts to commit it probes its local WCV for conflicts with remote transactions. If the WCV is empty (i.e., non-conflicting or read-only conflicting transactions), the core enters the commit phase. However, in case of conflict, the core enters the notification phase.

In this phase, the core sends abort messages (*AbortTx*) to all the cores marked in its WCV and waits for their response. A core that receives an abort request must check both its RCV and WCV to verify that there is a conflict with the committer. If so, the conflicting transaction is aborted and an *AbortAck* response is sent to the committer. Otherwise, the abort request is because of a conflict with a transaction that no longer executes on this core (either committed or aborted) and the request is ignored.

When all abort requests have been acknowledged, the notification phase ends. The core then enters the commit phase, where the core validates its data, so it becomes visible to the rest of the world. This is done by transitioning all cache lines accessed transactionally to a non-speculative state (T or W are moved to the M state and R lines are invalidated) and by clearing local signatures and Conflict Vectors. Unlike prior proposals, FUSETM does not brings additional communication with shared resources (*e.g.*, the L2 cache or the directory).

5.2.5.4 Local Aborts

In FUSETM, aborts are notified using core-to-core notification. When a core receives an abort notification (*i.e.*, a conflicting transaction is committing), it invalidates all the T, W and R lines from its L1 cache. Notice that FUSETM's aborts (like commits) do not require communication with the L2 cache or the directory.

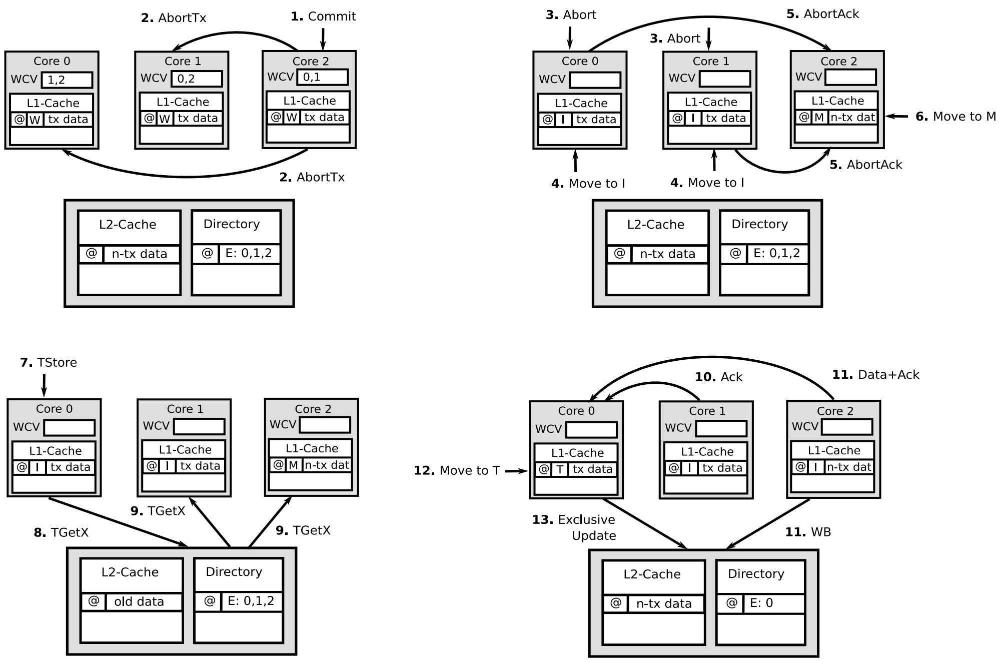


Figure 5.7: Retarded directory updates in FUSETM

FUSETM eliminates arbitration among lazy transactions, therefore two transactions may enter the notification phase at the same time. In order to prevent crossed conflicts, abort requests include a timestamp with the time a transaction started executing. When two transactions in the middle of their notification phase receive crossed abort requests, the younger transaction is aborted (it receives an *AbortNack* response).

In the uncommon case that two transactions report the same timestamp, the transaction executed on the core with a higher CPUid is aborted. Aborting committing transactions in their notification phase is safe to do because the memory state is not updated until a core enters its commit phase.

5.2.5.5 Retarded Directory Updates

In FUSETM, the directory is updated *lazily* by future remote requests, therefore committed lines may have multiple owners in the directory even though they may only exist in one L1 cache. Figure 5.7 shows an example of how FUSETM performs local commits and retarded directory updates. In the example, cores L_0, L_1 and L_2 are cores that have written line *A* inside a transac-

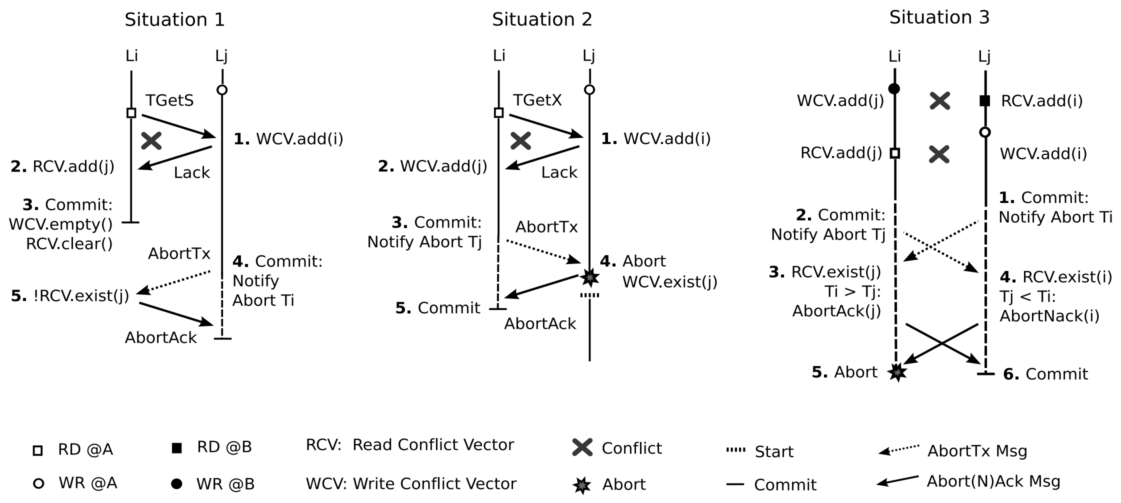


Figure 5.8: Local Commits and Abort Notification in FUSETM

tion. Eventually, L_2 commits (step 1), aborting L_0 and L_1 (step 2-4). After acknowledging the abort (step 5), L_2 becomes the only owner of line *A* because it holds the line in the Modified state (step 6), although the directory still has cores L_0 and L_1 as possible sharers.

When L_0 re-executes, it performs a *TStore* over line *A* and requests it from the directory (step 7). L_2 has not updated the directory at commit time, so the directory still maintains L_0, L_1 and L_2 as the owners of the line. Thus, the directory forwards the request to L_1 and L_2 (step 8 and 9). While L_1 acknowledges the request (it has invalidated line *A* during its abort, so it does not own the line anymore), L_2 sends the committed data to L_0 (step 10 and 11). After collecting all the responses, L_0 writes the line (step 12) and updates the directory by setting itself as as the exclusive owner of the line (step 13).

Similarly, if core L_0 performs a *TLoad*, the directory adds L_0 as a sharer in the directory (line *A* is kept in the S state in both L_0 and L_1 L1 caches).

5.2.6 Lazy Conflict Management in FUSETM

The prior section showed how lazy instructions deal with the memory subsystem individually. In this section, we describe how the entire lazy transactions are executed. Figure 5.8 shows how FUSETM executes lazy-mode transactions. In the following paragraphs, we describe how distinct situations are handled in our proposal.

Read Conflict (Situation 1): `Ti` is a read-only, lazy transaction that wants to load line *A*, which has been written by lazy-mode transaction `Tj`. After receiving the *TGetS* request, core `Lj` marks bit *i* in its WCV and replies with a *Lack* message (step 1). When core `Li` catches the message, it updates its RCV adding transaction `Tj` as a conflict (step 2). Eventually, `Ti` commits without conflict notification, cleaning the Conflict Vectors of core `Li` (step 3). (`Ti` eliminates the transactional state from caches, moving line *A* from `R` to `I`.) As it has its WCV empty, notification is not required. When transaction `Tj` commits, it checks its WCV and sends an *AbortTx* message to `Ti` (step 4). However, `Tj` does not appear in the RCV of `Ti`, so `Ti` acknowledges the request and continues its execution (step 5).
Write Conflict (Situation 2): Both transactions `Ti` and `Tj` write line *A* (step 1 and 2). Thus, they track the conflict in their WCV, requiring abort notification in their commit phase. At some point in time, `Ti` commits. Then, core `Li` checks its WCV and sends an *AbortTx* message to transaction `Tj`. Core `Lj` picks up the message, finds `Ti` in their WCV and aborts transaction `Tj`, replying the notification with an *AbortAck* reply (step 4). After that, `Ti` commits, transiting line *A* from the `W` to the `M` state.
Crossed Write Conflict (Situation 3): `Ti` and `Tj` are lazy-mode transactions with crossed conflicts (they have read and write lines *A* and *B*) that attempt to commit at the same time. Both transactions notify conflicts by sending abort messages (step 1 and 2), but only `Tj` successfully commits, because `Tj`'s timestamp is older than `Ti`'s (step 3 and 4). Both `Ti` and `Tj` wait until they collect all the replies from conflicting cores. `Tj` only receives *AbortAck* messages, therefore `Tj` moves to the commit phase, and *locally* commits the transaction (step 5). In contrast, `Ti` aborts as soon as it receives the *AbortNack* message from `Tj` (step 6).

5.2.7 Simultaneous Execution of Eager and Lazy Transactions

FuseTM executes eager-mode transactions when they do not fit on the L1 cache, but the system keeps running the rest of the transactions in the lazy execution mode. Eager and lazy transactions resolve memory inconsistencies using conventional strategies when they collide with same-mode transactions.

On the one hand, conflicts between eager transactions are resolved using the **Hybrid** (EE_{HP} in [14]) conflict management policy, which, as suggested in the results presented in Section 4.7.4,

	Eager Requester (Conflict)	Lazy Requester (Conflict)
Eager Receiver (Conflict)	Hybrid conflict resolution policy	Abort lazy requester (immediately)
Lazy Receiver Conflict	Abort lazy receiver (immediately)	Defer the resolution until commit time (comitter wins)

Table 5.1: Resolving eager-lazy conflicts in FUSETM

normally outperforms other conflict resolution policies. Note that this is a a high-performance policy that tries to avoid wasting computation by stalling transactions that issue conflicting requests. However, younger readers are aborted in order to minimize starvation for writing transactions. After aborting, an exponential backoff (based on the number of retries) is performed to guarantee the forward progress of eager transactions. On the other hand, lazy conflicts are resolved in a deferred fashion, aborting all the transactions that are inconsistent with the committer.

Nonetheless, in order to simultaneously execute transactions with different VM and CM schemes, FUSETM uses a conflict resolution policy that preserves the consistency of eager transactions and, at the same time, shields lazy transactions from eager modifications that have overflowed the cache. We have decided to implement a conflict resolution policy that prioritizes eager transactions over lazy transactions. Thus, this policy favors large transactions that overflow the L1 cache.

FUSETM implements an *eager-wins* policy when an eager transaction clashes with a lazy transaction. If the data is owned by the lazy transaction, the system aborts the lazy transaction and forwards to the eager requester the non-speculative value of the line from the L2 cache. Similarly, when a lazy transaction attempts to access to a memory location that belongs to an eager transaction, it automatically aborts its execution after receiving the *Nack* message. This strategy guarantees that the *large* eager transaction commits beforehand and, at the same time, prevents lazy transactions to read or write speculative overflowed data. Table 5.1 summarizes how FUSETM deals with conflicts while executing different-mode transactions.

5.3 A Speculative HTM System with Early Overflowing Updates

To the best of our knowledge, all HTM systems that postpone the resolution of conflicts for any kind of transaction implement late VM to store the overflowing speculative state [25, 45, 124] at the cost of adding expensive hardware. FUSETM eliminates this requirement by aborting the offending transaction that exceeds the L1 cache and restarting it in eager mode afterwards, using from that point ahead early VM.

Changing to eager mode on overflow introduces non-despreciable overhead that becomes critical when the system executes large transactions for two main reasons. First, lazy overflowing transactions discard useful work because they are aborted even in the absence of conflicts. Second, overflowing transactions must be re-executed from the very beginning in eager mode, which may restrict the overall concurrency of transactional threads.

In this section, we present SPECTM, a speculative HTM system that implements *early updates* on overflowing *lazy* data. This strategy allows SPECTM to keep speculating with most of transactional data in the presence of L1 cache overflows.

5.3.1 SPECTM Overview

SPECTM assumes similar infrastructure to FUSETM, being a fixed-policy, TCC-based HTM system built on top of a UTCP coherence protocol. It performs local commits together with core-to-core abort notification. However, it changes the L1 overflow engine to avoid inopportune aborts. Moreover, SPECTM keeps executing in lazy mode even when a transaction replaces speculative data—here the reason behind the name of the system.

On L1 cache replacements, SPECTM moves pre-transactional data to a software log and places new data on the shared levels of the memory hierarchy—SPECTM borrows the *selective logging* procedure implemented in FASTM-SL to accomplish this task. Evicting actions do not affect the behavior of overflowing transactions, which do not abort nor transit to eager mode. Thus, this technique allows lazy-mode transactions to move toward the commit point, deferring the resolution of most conflicts.

Using *early updates* on overflowing data presents non-trivial challenges for lazy HTM implementations. First, systems with lazy conflict management allow multiple versions of a line in distinct L1 caches. However, before replacing a cache line, the system must guarantee that the evicting core is the unique owner of that line, because that core is responsable of restoring its old state if the transaction aborts. Thus, an overflowing line must only have a single copy in the system.

Second, our approach moves overflowing data to the shared memory space, overwriting the old state kept in the L2 cache. As the old value of the line can no longer be obtained, the system must prevent remote transactions to access transactionally evicted lines, preserving those lines isolated from the world until the transaction commits or aborts—*i.e.*, eager resolution is needed for overflowed data.

SPECTM adds two novel mechanisms to achieve the above goals: *Partial Consistency* and *Overflow Isolation*. The next sections describe how these mechanisms operate as well as how they are implemented.

5.3.2 Partial Consistency

Like FUSETM, SPECTM holds non-conflicting transactional updates in the `T` state and conflicting written lines in a separate `W` state. Pre-transactional values of non-evicted cache lines are always kept in the L2 cache.

Before writing back the value of the transactionally modified cache line, the SPECTM must ensure that there are no live copies of the line in other private L1 caches. This is a straightforward step for consistent `T`-state lines, as they are exclusively owned by a single core. However, conflicting written lines—those lines that have been moved to the `W` state and thus potentially have multiple readers/writers in non-committed transactions—require additional actions to eliminate non-compatible values.

In order to invalidate all the transactional sharers of the cache line, the evicting core sends an *Abort* notification to all the cores that are present in its WCV, following exactly the same procedure as at commit time. When the remote cores receive that message, they abort, even if they have not touched the evicted cache line within their transactions. If an aborting transaction has already overflowed the L1 cache—*i.e.*, the software log is not empty—then the core must recover the old state using the procedure described in Section 4.6.

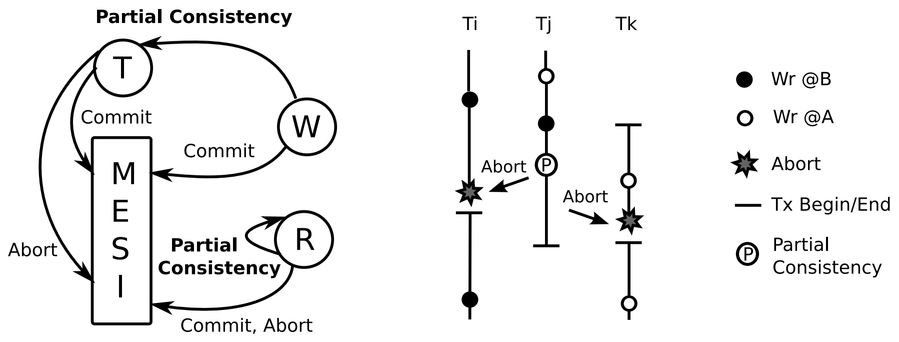


Figure 5.9: Partial consistency: Coherence transitions and early abort notification

After the abort process ends successfully, the evicting core is safe to write back the speculative value in the L2 cache. However, before that, the core (1) transits all the `W`-state lines to `T`, (2) clears the WCV list, (3) inserts the memory address of the evicted cache line in the Write Signature and (4) logs the pre-transactional data together with its physical addresses using the selective logging mechanism.

Partial Consistency guarantees that the replaced cache line has a unique owner in the system, as potential conflicters have aborted. What is more, it also guarantees that, at that point, *every* line being written inside the transaction only belongs to the overflowing transaction. Nonetheless, that transaction is not entirely isolated from the rest, as conflicting read lines—those kept in the `R` state—may still be found in the write set of other in-flight transactions. If those conflicts still remain at the end of the transaction, the committing transaction will abort the readers to maintain the consistency of the system.

The leftmost diagram of Figure 5.9 shows how Partial Consistency modifies the transitions of the UTCP Coherence Protocol. The rightmost picture of Figure 5.9 shows how transaction `Tj` performs *Partial Consistency* before evicting block *B*. `Tj` must abort `Ti` because both transactions have written line *A* before in the transaction—SpecTM can only evict consistent data, so the system must enforce that `Tj` does not have a conflict with any in-flight transaction.

5.3.3 Overflow Isolation

In SpecTM, overflowing data must be preserved in isolation—no in-flight transaction can access that data until the transaction commits or aborts. To guarantee that invariant, when a core receives a coherence request from a remote transaction, it checks its Write Signature.

If the address is present in the filter, then the core replies with an *Abort* message, and the remote transaction aborts immediately. The right picture of Figure 5.9 shows how transaction `Tj` prevents the access of L1 evicted block *A* when transaction `Tk` attempts to write it.

Note that overflowing transactions may produce cascades of aborts (either when there are continuous evictions of transactional data or when overflowing transactions access speculative data that has been moved out of the L1 cache). As these transactions have to be recovered by software, we decided to perform a randomized exponential backoff before restarting the transaction. This strategy reduces contention to ensure forward progress in the application.

For exemplifying how overflowing data is kept in isolation after Partial Consistency, we assume that transactions `Ti`, `Tj` and `Tk` have written line *A*. As can be seen in Figure 5.10, the transaction executed in core `Ci` wants to replace line *A* (step 1). This causes the abort of transactions `Tj` and `Tk` (step 2). Then, `Ci` cleans its WCV vector, adds line *A* in the Write signature (step 3), brings the old value of line *A* from the L2 cache (and takes the physical address of the line from the L1 tag) and inserts a new entry on top of the software log (step 4). After that, the system can safely write back the speculative value in the L2 cache and set core `Ci` as the exclusive owner of the line (step 5). Assume now that transaction `Tj` attempts to write the evicted line *A* (step 6). The directory forwards the request to `Ci` (steps 7 and 8), who denies the access to preserve the isolation of the overflowed cache block because it finds a match in its Write Signature (step 9). Consequently, `Ci` replies aborting transaction `Tj` (step 10).

5.3.4 Coherence States: Codification and Implementation

Figure 5.11 presents a possible codification for the UTCP coherence states. We augment a classical MESI layout with two additional bits: the transactional (T) bit—used for conflicting read data (`R` state) and transactionally written data (`W` and `T` states)—and the speculative (S) bit—used only for conflicting written data (`W` state).

Figure 5.11 also shows a six-transistor SRAM cell implementation of the above codification. Each cell keeps a bit, and the system is extended with circuitry for flash-clearing the transactional state. Each bit uses a different signals for changing the cache line state in parallel. These signals use three in-core flags to control the logic of the circuitry, which are set when

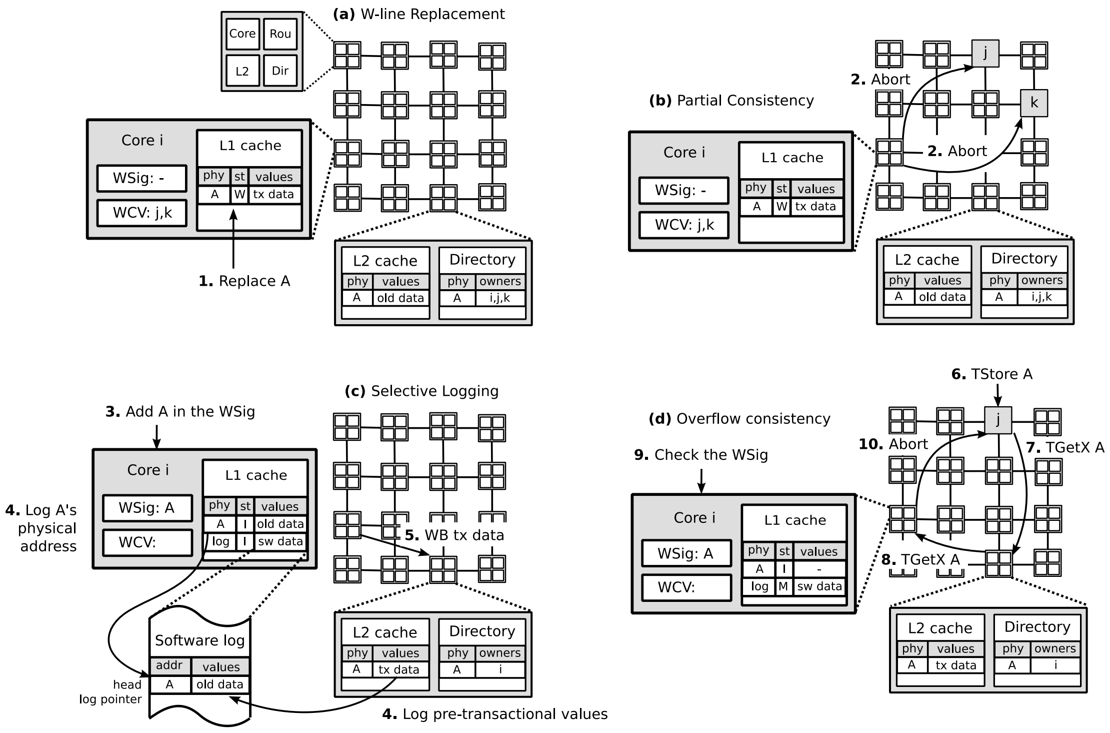


Figure 5.10: Unbounded hardware support for partial consistency (a,b), selective logging (c) and overflow isolation (d) in SPECTM

(i) a `W` line is evicted from the L1 cache (Partial Consistency, `PCon` for short), (ii) a transaction receives an *AbortTx* message (`Abort`) or (iii) a transaction reaches its end (`Commit`).

After triggering *Partial Consistency* and collecting all the *AbortAck* messages, all `W`-state lines must transit to `T`. Our implementation accomplishes that by activating the W_{clear} signal when the `PCon` flag is set, which flash-clears the S bit of each `W`-state line simultaneously.

Similarly, `W`-state and `T`-state lines must be moved to Modified in the commit phase—this happens when the `Commit` flag is set to one. Thus, after the notification phase, the S and T bits are pulled down to zero by asserting signals W_{clear} and T_{clear}. This is also straight for `R` lines, which must transit to Invalid. Note that the `R` state does not have the Dirty (D) or Valid (V) bit activated, so the transition is immediate.

If the `Abort` flag is asserted, the core turns on the W_{clear}, the T_{clear} and the D_{clear} signal to invalidate all transactional lines. Before clearing the S and T bits, transactionally modified lines must pull down to zero the D bit. (The D bit is already zero for `R`-state lines.) This only happens

Coherence Protocol States

State	S	T	D	V
W	1	1	1	0
T	0	1	1	0
R	0	1	0	0
M	0	0	1	0
E	0	0	1	1
S	0	0	0	1
I	0	0	0	0

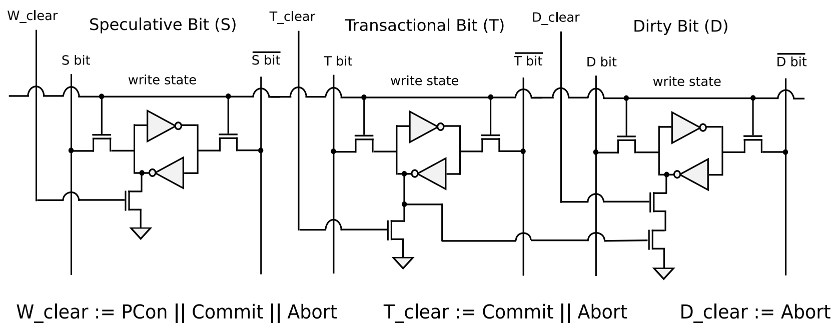


Figure 5.11: Codification and implementation of cache coherence states

for those L1 cache entries where the T bit is one—this ensures that non-transactional lines are not flushed from the L1 cache. After that, S and T bits are flash-cleared.

5.4 Evaluation

For our analysis we have chosen to compare FUSETM and SPECTM with different lazy (TCC-like) HTM systems that utilize an idealized late VM for overflowing data. These implementations, which serve as upper-bounds, never log values in software; instead they keep transactionally evicted lines in an infinite victim cache. This cache has the same latency as the L1 cache for reads and writes. The transactional victim cache moves committed values to the L1 instantaneously, and it has a zero-cost abort recovery—transactional entries are just discarded.

The idealized lazy HTM systems employ distinct commit protocols. Thus, we compare our *local* approach with two other techniques proposed for modern TCC-based HTM systems. All the proposals use the UTCP protocol for keeping the speculative state, 32-bit RCV and WCV and core-to-core abort notification. The rest of the parameters used in the evaluation are the ones described in Chapter 3. Following is a description of the parallel commit protocols analyzed in the evaluation for idealized lazy HTM systems.

Distributed Commit [92] (TCC-Dist): This is protocol used in the lazy reference system described in Section 3.2.2. The system acquires the directory modules accessed during the transaction before making transactional writes globally visible. Hence, transactions that modify different directories can commit in parallel. Acquired directories are blocked only for commit purposes—the directory modules can still forward memory requests. Directory steals are allowed to eliminate deadlocks.

Selective Commits [124] (TCC-Sel): This commit protocol only acquires/releases directory modules that possess a conflicting line. Like in EazyHTM [124], TCC-Sel assumes that non-conflicting lines are exclusively owned by the committing transaction (the directory is up-to-date for those lines, so it is not necessary to update the module). However, entries that record conflicting lines must be atomically acquired (and after that, updated) to keep the system consistent. Thus, selective commits permits a truly parallel commit on non-conflicting (or read-only conflicting) transactions, but arbitration and directory updates remain for those with inconsistencies.

Local Commits (TCC-Loc): This is the mechanism proposed for FUSETM and SPECTM. Like in SPECTM , all transactions are executed in lazy mode—the system does not abort overflowing transactions to restart them in eager mode, as FUSETM does. However, it does not implement partial consistency nor overflow isolation, given that speculative L1 evicted data is held in an unbounded victim cache. Note that TCC-Loc performs local commits. As a result, the system eliminates directory communication from the commit phase.

5.4.1 FUSETM Performance Analysis

Figure 5.12 presents the time distribution of TCC-Dist (labeled D), FUSETM (labeled F) and TCC-Loc (labeled L) HTM systems in their 32-threaded executions (for low-contention applications, top of Figure 5.12) and in their 16-threaded executions (for high-contention[2], bottom of Figure 5.12) applications.

The execution time has been normalized to the 32-threaded (low-contention) and 16-threaded (high-contention) TCC-Dist execution and is broken down to: non-transactional and barrier cycles (labeled Non-Tx and Barrier), the time spent in committed transactions (labeled Good Tx), the time that is wasted in non-useful work discarded from aborted transactions (labeled Aborted Tx), the time spent in abort recovery and in the commit phase (labeled Aborting and Commit), the time that transactions remain stalled waiting for a conflict to be resolved (labeled Stalled), and the time that processors execute the exponential backoff after aborting (labeled Backoff).

[2]we refer as high-contention applications those that presented high abort rates in Section 3.3.2. whereas low-contention are those that present low and medium abort rates

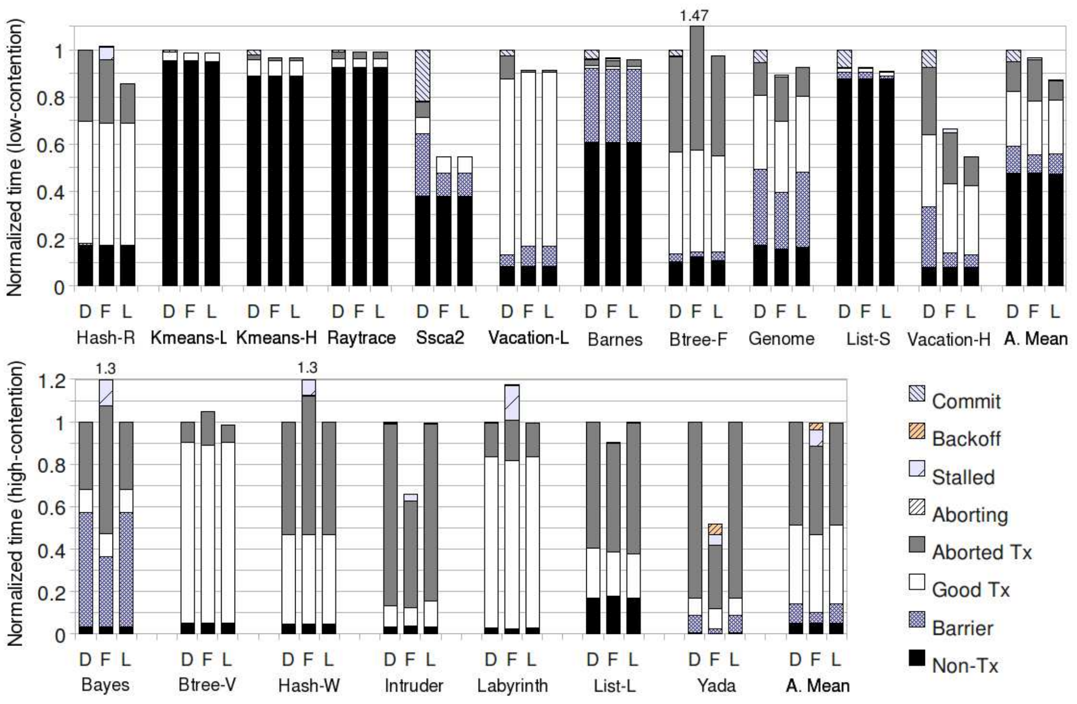


Figure 5.12: Distributed executed time of low- and medium-contention (top, 32 threads) and high-contention (bottom, 16 threads) TM applications under TCC-Dist (D), FUSETM (F) and TCC-Loc (L) HTM systems

Note that Stall and Backoff cycles only appear in the FUSETM bar, given that this is the only system that executes eager transactions.

As it can be seen in Figure 5.12, FUSETM performs close to both idealized VM systems, although it does not require additional hardware to pin down overflowing data. In fact, FUSETM outperforms TCC-Dist by a 3% in low-contention applications, behaving like TCC-Loc in most of the workloads. Nonetheless, FUSETM is a step far from TCC-Loc in applications with large benchmarks, like *Btree-fix* or *Hash-read*.

In high-contention applications, FUSETM also presents pretty good performance. In average, it only slows down around a 0.5% with respect to the idealized lazy HTM systems. This is because the elegant overflow policy that FUSETM incorporates: by re-executing overflowing transactions in eager mode, the system reduces to the bare minimum the buffering support without scarifying performance. Only *Bayes* and *Hash-write* downgrade their performance under FUSETM due to the overheads produced by eager transactions. In contrast, other applications

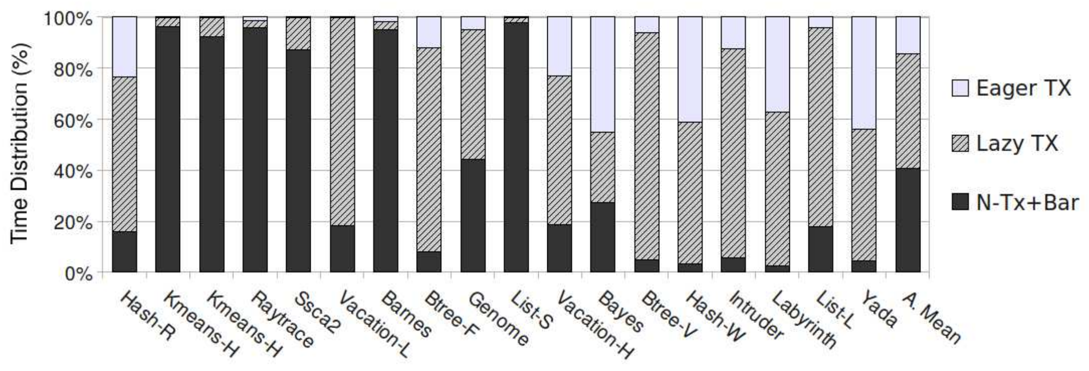


Figure 5.13: Normalized FuseTM execution time of applications distributed by the transactional mode

like *Intruder* or *Yada* experiment the opposite behavior. The reasons for such performance are summarized in the following paragraphs.

High-performance dual-mode execution. FuseTM successfully integrates lazy execution with eager-like hardware support. Consequently, both modes of execution exhibit good (and comparable) performance (more on this in Section 5.4.4). What is more, FuseTM successfully combines eager and lazy transactions when cache overflows occur. Figure 5.13 breaks down FuseTM HTM execution time to the time spent in non-transactional code or barriers (labeled N-Tx+Bar), the time spent in eager (overflowed) transactions (labeled Eager TX) and the time spent in lazy transactions (labeled Lazy TX). As it can be seen, in applications that execute small and large transactions (*e.g.*, *Genome*, *Intruder*, *List-long*), FuseTM prioritizes eager-mode transactions while keeping lazy-mode transactions under fast execution.

Fast lazy commits. FuseTM performs *local* commits, a technique that eliminates the communication with shared resources at commit time. This is especially helpful in applications with read-only transactions, like *Hash-read* or *Vacation-high*, or in applications with tiny-size transactions, like *List-short* or *Ssca2*. As can be seen in Figure 5.12, the time spent in local commits (either in FuseTM or in TCC-Loc) is inconsequential, which reports up to a 80% speedup over state-of-the-art committing techniques in applications with small read/write sets.

Efficient unbounded transactions. Our evaluation shows that overflows are common. Albeit the important overheads of discarding more transactional work (around 2X over TCC-Loc in

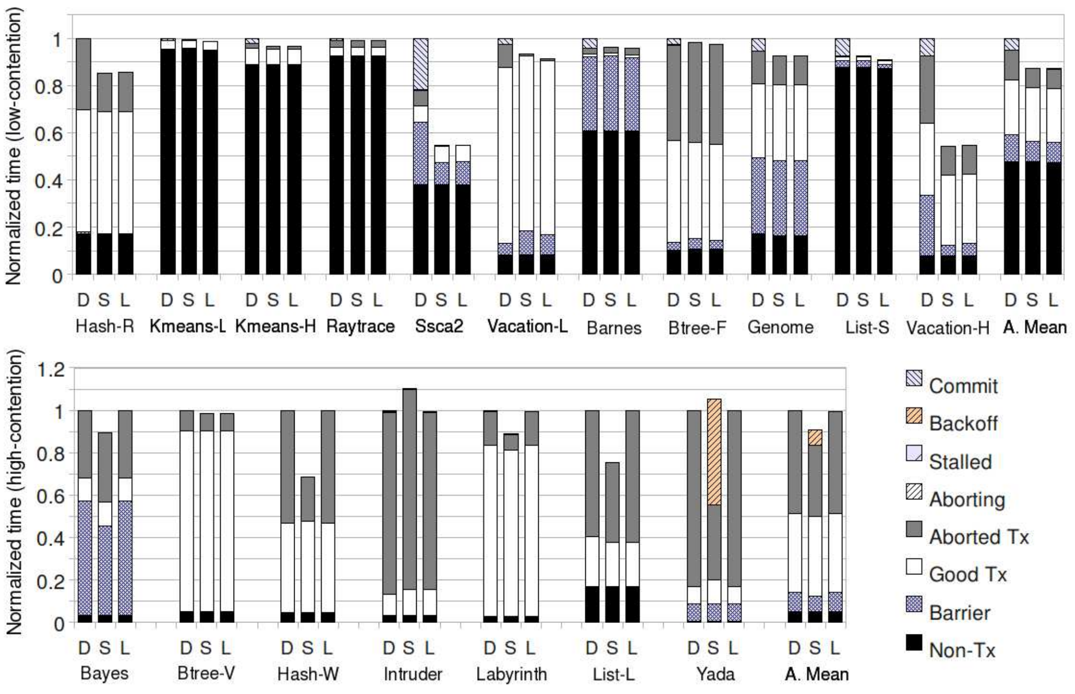


Figure 5.14: Speedup achieved in low-contention (top, 32 threads) and high-contention (bottom, 16 threads) applications in TCC-Dist (D), SpecTM (S) and TCC-Loc (L)

high-contention applications) and introducing a randomized backoff, switching to eager mode on overflows in applications like *List-long* or *Yada* reduces the impact of the *starvation of the elder* pathology [14]. Thus, in some occasions, FuseTM is better than idealized, fixed-policy HTM systems, with the added benefit of lower hardware cost.

5.4.2 SpecTM Performance Analysis

Figure 5.14 shows the execution time of SpecTM (labeled S) compared with the normalized execution of TCC-Dist (baseline, labeled D) and the TCC-Loc (labeled L) systems. Figure 5.14 groups execution cycles using the same classification as Figure 5.12.

As it is shown in Figure 5.14, SpecTM experiments a similar behavior to TCC-Loc using bounded hardware support, and much better performance than TCC-Dist because it implements local commits. In average, it only performs a 0.4% worse than TCC-Loc in low-contention applications. Moreover, SpecTM yields better performance than TCC-based implementations in applications with large transactions and conflicts, such as *Hash-write* or *List-long* because of the reasons listed below. In average, SPECTM reports a 10% speedup over TCC-Loc in high-contention applications.

Efficient late data versioning. SPECTM offers an elegant data versioning mechanism that combines the benefits of selective logging for overflowed data (early data versioning) with the smartness of handling few multiple versions of conflicting lines in the L1 cache (late data versioning). This mechanism extends lazy resolution of conflicts even for those applications that overflow finite data versioning buffers, which permits more concurrency than eager solutions when transactions collide, eliminate some read-write violations and remove backoff in all the cases but *Yada*, where multiple overflowing transactions access L1 evicted data,
Anticipated resolution of conflicts. SPECTM implements partial consistency when a conflicting transaction evicts transactional data, solving the inconsistencies among transactions before commit time. Moreover, keeping the isolation of overflowed speculative values also guards long-standing modifications against younger writes. This permits SPECTM to outperform TCC-Loc by a 10% in benchmarks like *Bayes*, *Labyrinth* or *List-long*—and up to a 30% in *Hash-write*. Nonetheless, the impossibility of adaptation at runtime produces a negative effect in environments like *Intruder* or *Yada*, which are far from the performance exhibited by FUSETM.

5.4.3 Local Commit Analysis

Both FUSETM and SPECTM perform *local* commits, a technique that eliminates the communication with shared resources at commit time. This is helpful in low-contention applications, especially if they execute small transcations. In fact, the data on Figure 5.13 report that 5% of the execution time in low-contention workloads is spent in commits when a non-optimized protocol is used. Instead, our mechanism requires less than 0.1% of the execution time on commits. In high-contention, coarse-grained applications the overhead is not so critical because most applications hide the commit latency by computing large transactional chunks.

Nonetheless, in environments that execute small transactions like *List-short* or *Ssca2* the usage of local commits report significant performance benefit, improving up to an 80% the performance of the distributed commit implementation. In medium-contention applications like *Vacation-high* the usage of local commits also reduces the conflict window (fewer aborts

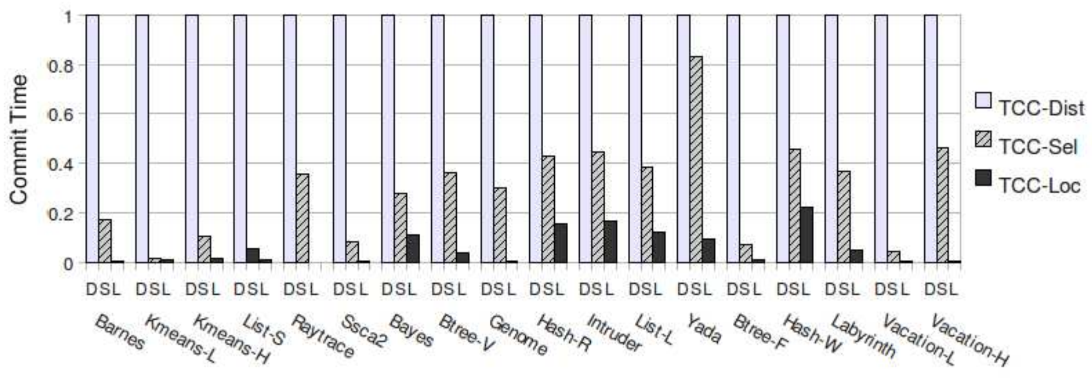


Figure 5.15: Normalized commit time under 32-threaded TCC-Dist (D), TCC-Sel (S) and TCC-Loc (L)

are notified), which permits a more balanced execution of transactional threads (fewer Barrier cycles).

Figure 5.15 presents the time spent on commits in three idealized VM systems that implement distinct commit protocols: TCC-Dist (labeled D), TCC-Sel (labeled S) and TCC-Loc (labeled L). Commit time is normalized to TCC-Dist and applications are executed under 32 threads. Figure 5.15 shows that *local* commits accelerate the commit phase of the distributed (selective) approach by an average factor of 20X (4X). The improvement is even more notable in applications with non-conflicting transactions (*e.g.*, *Kmeans-low* or *Ssca2*), where selective and local commit protocols almost reduce to zero the commit impact. Nonetheless, in high-contention applications only our technique is able to reduce the commit overhead (in *Vacation-high* up to 100X over distributed, up to 60X over selective).

Figure 5.16 shows the average number of network messages per transactions introduced in the commit phase by the three idealized VM systems. Although fine-grained applications have small read and write sets, they introduce lots of messages in the network at commit time under TC-Dist. The main reason is that most of the transactions fail in their attempt of acquiring directory modules, so they require several retries (an even directory steals) before acquiring a single module. This behavior can be observed in *Barnes*, *Raytrace* or *Ssca2*. Similarly, TCC-Sel suffers long delays in high-contention applications like *Btree-var* or *Yada* because transactions must acquire conflicting directories before committing. In contrast, TCC-Loc only needs to

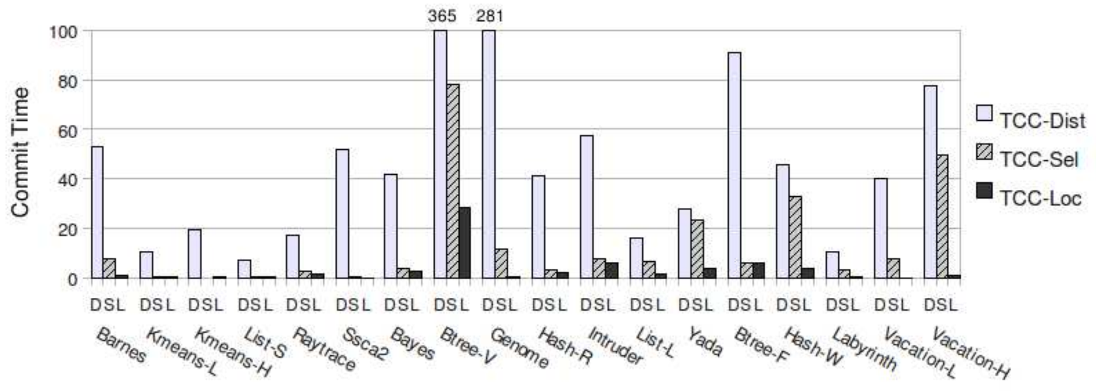


Figure 5.16: Average network messages in the commit phase under 32-threaded TCC-Dist (D), TCC-Sel (S) and TCC-Loc (L)

notify aborts (a point-to-point message) for conflicting transaction, minimizing the number of network messages delivered on commits.

Note that the commit protocols recreated in this dissertation only take into account the time spent in commit arbitration, given that data transfers and directory updates are not required in our framework. Nonetheless, other lazy HTM systems impose data broadcast and extra communication at commit time [20, 45, 124], which increases the delay produced at the end of the transaction.

5.4.4 Eager and Lazy Execution Analysis

Figure 5.17 presents the time distribution of two high-performance fixed-policy HTM systems —an eager HTM system (FASTM with the *Hybrid* resolution policy, labeled E), and a lazy HTM system (the idealized TCC-Loc, labeled L) in their 32-threaded executions (for low-contention applications, top of Figure 5.17) and in their 16-threaded executions (for high-contention bottom of Figure 5.17) applications. The distribution time has been normalized to FASTM execution and has broken down using the same categories as Figure 5.12.

As it can be seen in Figure 5.17, lazy-mode transactions outperform by a 14% eager-mode transactions in low-contention applications, while in high-contention applications eager-mode transactions obtains better performance (FASTM gets a 19% speedup over TCC-Loc). We explain the reasons why the three HTM systems achieve such accomplishments in the following paragraphs.

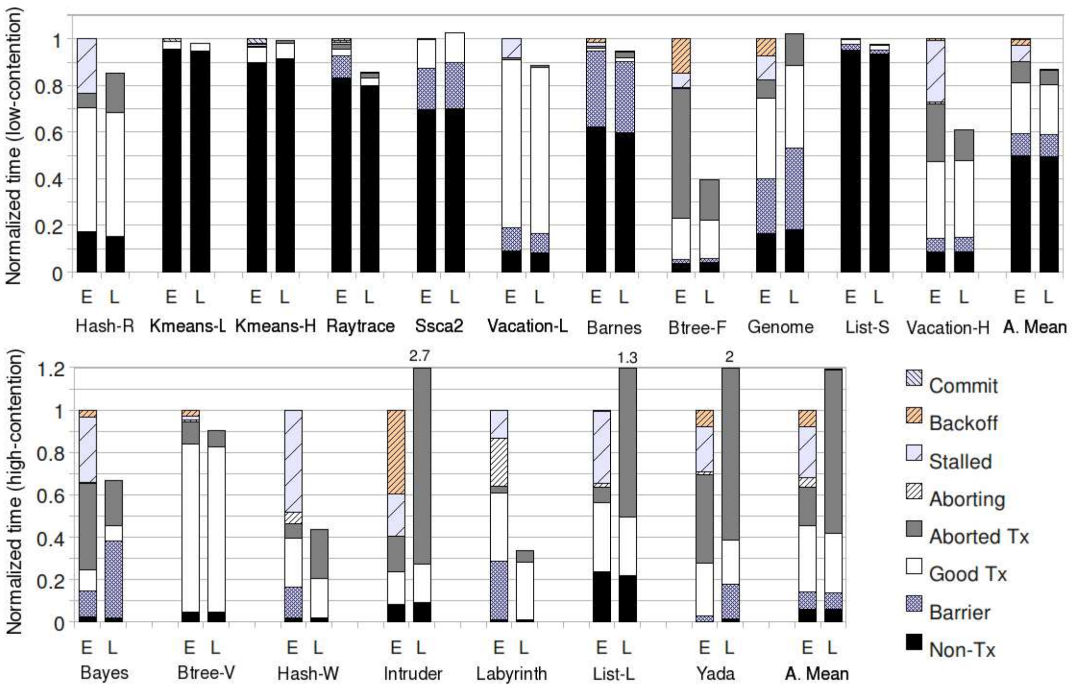


Figure 5.17: Distributed executed time of low- and medium-contention (top, 32 threads) high-contention (bottom, 16 threads) TM applications under Eager FASTM (E) and Lazy TCC-Loc (L) HTM systems

Eager HTM weaknesses. Eager HTM systems systems—even the FASTM that implements high-performance conflict and version management policies—are not effective when collisions among threads are frequent. In FASTM execution, transactions are stalled in the case of conflict, which may lead to futile stalls (transactions that abort after being stalled for a long time) or cascades of stalls (transactions that are stalled by transactions that have been already stalled, which are waiting for other conflicts to be resolved). This behavior typically occurs in many-threaded applications with read-write conflicts, like *Hash-write* or *Vacation-high*, or in applications with long transactions like *Labyrinth*. On average, 27% (6%) of the eager high-contention (low-contention) execution time is spent in stalled transactions. Moreover, eager transactions utilize an exponential backoff that is based on the number of retries to spread the computation and avoid livelocks. Backoff is critical in high-contention applications with large transactions, like *Btree-fix* or *Yada*, or in applications with lots of aborts and small transactions, like *Intruder* or *Genome*. On average, 8% (2%) of the high-contention (low-contention) eager execution time is spent in the randomized backoff.

Lazy HTM weaknesses. Lazy HTM systems may abort older writers several times, which results to an important amount of discarded transactional work (TCC-Loc wastes 4X more than FASTM execution in high-contention benchmarks). This is critical in applications with large transactions, like *Intruder* or *Yada*. However, applications with small transactions and read-after-write conflicts, such as *Btree-fix* or *Hash-write*, improve their performance over the eager baseline. This performance improvement is due to (i) the speculative resolution policy that TCC-Loc employs—which do not stall conflicting memory accesses nor require backoff—and (ii) the efficiency of *local* commits—which drop off the time spent in the commit operation.

5.4.5 FUSETM and SPECTM Execution Analysis

Figure 5.18 shows the execution time of both proposals (FUSETM , labeled F, and SPECTM , labeled S) using the normalized execution of TCC-Loc (labeled L) as a pure, high-performing lazy baseline. Figure 5.18 classifies the execution cycles according to the previous standards, using 32 threads (low-contention) and 16 threads (high-contention).

In average, SPECTM obtains less than a 1% performance degradation in low-contention benchmarks compared with the ideal TCC-Loc, while in high-contention applications it beats the baseline by a 10%. On the other hand, FUSETM achieves an 8% (low-contention) and a 0.5% (high-performance) of performance degradation with respect to TCC-Loc, albeit it requires less hardware extensions than SPECTM. The following paragraphs summarize the upsides and downsides of both speculative HTM systems with local commits.

FUSETM strengths. By re-executing overflowing transactions in eager mode, FUSETM obtains similar performance to FASTM in coarse-grained applications, as it can take advantage of eager conflict management for large transactions. This fact reduces the number of aborts in *Intruder* or *Yada*, given that, after receiving a conflict notification, the system is able to preserve useful computation from large transactions. On these benchmarks, FUSETM beats SPECTM by a factor of 2X. Moreover, FUSETM can rely on conventional logging mechanisms, being less hardware-invasive than SPECTM.

FUSETM weaknesses. FUSETM must abort the whole overflowing transaction, which incurs in an increment of Aborted Tx cycles in low-contention applications with long transactions

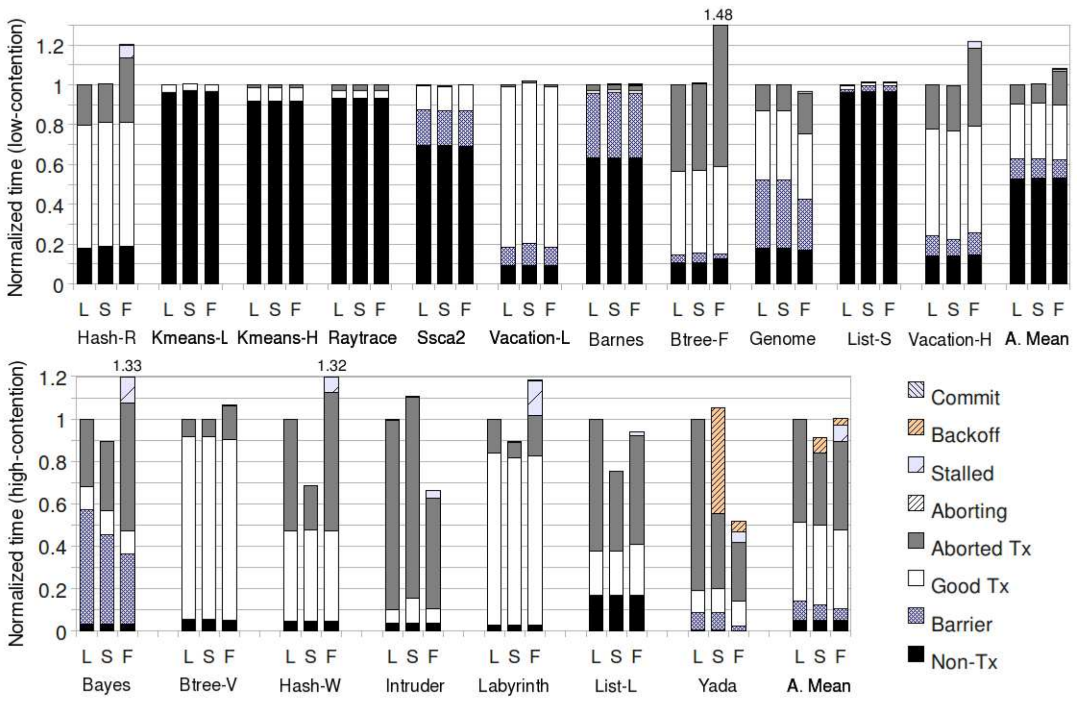


Figure 5.18: Distributed executed time of low- and medium-contention (top, 32 threads) high-contention (bottom, 16 threads) TM applications under TCC-Loc (L), FUSETM (F) and SPECTM (S) HTM systems

like *Hash-read* or *Btree-fix*. Moreover, it forces eager execution for overflowing transactions in medium- or high-contention applications, which is not the best alternative in *Bayes* or *Vacation-high*. On some applications, these disadvantages report up to a 50% performance degradation with respect SPECTM.

SPECTM strengths. SPECTM does not suffer from FUSETM's limitations described above. It resolves most of the conflicts lazily, independently of the transactional size. This is preferable in applications like *Btree-var* or *Labyrinth*, where read-write conflicts are common. Moreover, it avoids aborts of transactions that overflow at the cost of *early* abort notification. Resolving some conflicts beforehand provides additional benefit over TCC-Loc in benchmarks like *Bayes*, *Hash-write* or *List-long*. Overall, SPECTM obtains a 9% speedup over FUSETM in high-contention applications, and around an 8% in low-contention applications.

SPECTM weaknesses. As pointed out in the last section, lazy conflict management sometimes performs worse than eager conflict management. In those ocassions, SPECTM will perform similar to TCC-Loc, which is far from the performance of an eager HTM system such as FasTM. Nonetheless, FuseTM re-executes large transactions in eager mode, incorporating by default a better resolution of conflicts in *Intruder* or *Yada*. In fact, SpecTM spends almost half of *Yada*'s execution cycles in backoff. This is because lots of conflicts involve transactional evicted lines. Nonetheless, FuseTM eliminates this problem by stalling the transaction on conflict.

5.5 Related Work on Lazy HTM System

Transactional Coherence and Consistency (TCC [45]) proposed to decompose parallel executions on chunks of computation to reorder at free will memory accesses. In later refinements of the system, those chunks were defined by the programmer using transactional semantics. Bulk [21] used a similar approach to implement efficient sequential consistency. Both TCC-like and Bulk-like HTM implementations require some kind of global arbitration—involving either token acquisition or bus blocking. This process makes committing data one of the principal bottlenecks of lazy HTM systems. Hence, distinct proposals have emerged to reduce the commit overhead, allowing distributed arbitration and memory updates to increase the scalability of TM applications.

Chafi *et al.* employed a directory-based coherence protocol to increment the scalability of a TCC environment (ScalableTCC [22]). Processors must acquire a global *ticket* from a centralized agent, and then acquire (block) each directory bank being read or written in the committing transaction. Probe messages are sent to check if transactions with smaller tickets have acquired their writing directories, stalling younger committers if they do not. After all directories are acquired, each directory entry of the write set has to be updated, setting the committer as the new owner of the line. Of course, invalidate messages are sent to the concurrent sharers/owners, which force the abort of conflicting transactions.

Pugley *et al.* proposed different protocols to eliminate the use of a centralized agent that delivers tickets [92]. In their SEQ-TS strategy, distributed directory banks are acquired in parallel, recording in that moment a timestamp that informs at which moment the transaction has started its commit phase. To avoid deadlocks, older committers must steal younger directory acquisitions by sending a specific message. If so, younger committers release stolen directories and restart the commit process again. Once a processor has acquired all the directories, it asserts

HTM System	VM Strategy	Hardware Support	Conflict Detection	Overflow Policy	Commit Process	Arbitration Mechanism
TCC [45]	Late	L1 TX cache	Lazy	Serialize long Tx	Broadcast Rd & Wr set	Centralized arbiter
ScalTCC [22]	Late	L1 TX cache	Lazy	XTM [25]	Serialize dir. acquisition	Centralized TID delivery
SEQ-TS [92]	Late	L1 TX cache	Lazy	No info	Parallel dir. acquisition	Block dir. modules
Bulk HTM [20]	Late	L1 TX cache	Lazy	LTM [5]	Broadcast signatures	Acquire bus
Eazy HTM [124]	Late	L1 TX cache	Eager	Victim cache	Update directories	Block dir. entries
FlexTM [112]	Late	Coherence protocol	Eager	Overflow Table	Software notification	Software
FUSETM	Late or Early (OV)	Coherence protocol	Eager	Switch to Eager Tx	Local commits	-
SPECTM	Mostly Late	Coherence protocol	Eager	Overflow Isolation	Local commits	-

Table 5.2: Characteristics of lazy HTM implementations

a local flag to prevent future steals, and then starts updating all the directory entries in the write set, not unlike ScalableTCC.

EazyHTM [124] uses eager conflict detection to track violations among transactions, but defer the resolution of conflicts until commit time, just like FlexTM [112] does. In EazyHTM, transactions without conflicts can commit in parallel, serially writing the transactional values in the shared memory. If a conflict exists, the committing core must notify aborts and then proceed with directory updates, acquiring exclusive permissions for each line contained in the write set, which in turn invalidates copies kept in other processors.

Recent Bulk CMP implementations also optimize the chunck commit phase. BulkSC [21] adds a centralized arbiter that takes each of the write signatures of the already committing chunks and intersects them with the read/write signature of the incoming committer. If the intersection is empty, parallel commits are allowed. A distributed arbiter can also be used to improve the scalability of the system. In BulkSC, signature expansion is performed in the directories to avoid massive broadcast of signatures. ScalableBulk [93] increases the overlap between committing chunks by grouping directory banks, blocking accesses to a set of shared cache lines and performing signature disambiguation in directory modules.

Table 5.2 summarizes the main characteristics of existing lazy HTM systems. As it can be observed, both FUSETM and SPECTM use a novel version management scheme for keeping the speculative state, with results in improved commits and no arbitration. Both schemes also simplify the overflow hardware compared with other lazy HTM systems: FUSETM relies on eager transactions to accomplish that task whereas SPECTM takes advantage of a co-designed conflict and buffering engine. Most importantly, both solutions fit in a conventional CMP framework, being less intrusive that previous lazy HTM systems.

5.6 Conclusions

In this chapter, we have presented FUSETM and SPECTM, the first lazy HTM systems that entirely remove global arbitration and communication with shared resources at the end of transactions to provide fast commits. Both of them extend a typical coherence protocol with two additional solid L1 states to separate conflicting and consistent memory blocks inside the first level cache, which permits an easy identification of non-isolated data. These system also track in per-core bit vectors conflicting transactions, use core-to-core abort notification and postpone directory updates to perform local commits.

FUSETM is the first HTM system that permits simultaneous execution of eager- and lazy-mode transactions. FUSETM offers high-performance lazy execution for those transactions that fit in the L1 cache. In the case of cache overflow, the system aborts the offending transaction and re-executes it in eager mode, which uses early data versioning with logging support. This approach is much simpler and efficient than previous approaches, which assumed late data versioning for the overflowed state.

FUSETM substantially reduces the hardware cost compared to prior lazy HTM designs without loosing performance. Evaluation results show that FUSETM obtains close performance to an idealized HTM system that does not suffer from transactional cache evictions. In some applications FUSETM even outperforms the idealized approach because eager conflict management can be more effective when dealing with large transactions. The evaluation also demonstrates the effectiveness of using local commits in fine-grained applications. Our mechanism achieves up to 80% speedup over other commit protocols that recently appeared in the literature. In fact, our technique removes the commit phase from the critical path—less than 0.1% of the execution time is spent on commits.

FUSETM forces overflowing transactions to be re-executed in eager mode. We have seen that this procedure generates extra aborts, and limits the concurrency obtained by pure lazy HTM systems in some applications. SPECTM proposes a realistic VM scheme to maintain isolated speculative overflowing data in the upper levels of the memory hierarchy, while in-cache values allow inconsistencies. Therefore, SPECTM recreates a truly lazy HTM system where most of the conflicts are resolved at commit time. This approach obtains a 17% speedup over FUSETM in high-contention applications.

Finally, we offer the most complete evaluation of both eager and lazy HTM designs using the same simulation infrastructure, comparable HTM support and hardware configuration. This methodology permits a more meaningful comparison of the various HTM systems. We show that lazy schemes are more efficient when dealing with small transactions with high-contention, because they guarantee forward progress without requiring backoff schemes and they can speculate on read-write conflicts. In contrast, eager schemes are better suited for large transactions, because they implement simpler overflow policies and can preserve transactional computation in the case of conflict by stalling the uncompleted request.

Chapter 6

High-Performance Adaptive Hardware Transactional Memory Systems

Most Hardware Transactional Memory implementations choose fixed version management (VM) and conflict management (CM) policies at design time. Although there are few exceptions (see related work in Section 6.6), most HTM implementations fit in one of two categories: they are either eager or lazy[1] HTM systems.

On the one hand, eager HTM systems present poor performance when they execute applications with a high number of conflicts. In these scenarios, eager HTM designs may abort a transaction multiple times before it commits [14]. Moreover, most implementations require a backoff policy to avoid repetitive conflicts between aborting transactions [98]. Nonetheless, eager approaches can preserve the computation generated on large transactions by stalling conflicting requesters, and they do not require additional commit actions [112].

On the other hand, lazy HTM systems suffer considerable delays when hardware resources are overflowed—*e.g.*, in FUSETM, large transactions discard transactional computation on overflows because the system must re-execute them in eager mode. Moreover, lazy solutions must abort all the transactions that conflict with the committer, which may result to starvation of older transactions [109] and may increase the amount of transactional wasted work [112]. Nevertheless, lazy HTM systems can avoid some read-write conflicts and guarantee forward progress

[1]in this chapter, we refer as eager HTM systems those that implement eager CM, whereas lazy HTM systems implement lazy CM, independently of the VM strategy that they use.

without applying a backoff policy. Therefore, lazy approaches are more effective when they deal with small, high-contention transactions [14].

Experiments presented in the previous two chapters show that each scheme has its strengths and weaknesses. Nonetheless, both approaches lack flexibility when they resolve non-trivial conflicts on different-length transactions. In this chapter, we present two fully-flexible HTM systems that can adapt the hidden transactional mechanisms according to the size and contention of any instance of a transaction executed in the system. The former proposal is DYNTM, the first truly adaptive HTM system that implements a novel conflict resolution policy between distinct-mode transactions and utilizes a simple predictor to decide the best execution mode for each transaction at runtime. The latter proposal is SWAPTM, an effective alternative to DYNTM. SWAPTM records in hardware important statistics of the activity of the in-flight transaction to interchange the conflict management strategy without additional actions. Because of SWAPTM decouples the VM strategy from the CM policy, transactions' mode of execution is not subjected to their length.

This chapter is organized as follows. First, it motivates the work for DYNTM and SWAPTM, showing the main limitations of fixed HTM systems. Then, it overviews the DYNTM system, describing the programming model, the hardware required to determine the best-suited execution mode for each instance of a transaction and the proposed conflict management policy. After that, the chapter reveals how SWAPTM can take advantage of SPECTM's VM strategy to get rid of DYNTM's predictor and use instead profiling information of the current instance of a transaction to determine the most profitable mode of execution. The chapter follows evaluating DYNTM and SWAPTM , and presenting a global vision of how this thesis contributes to improve the quality of HTM state-of-the-art. The chapters ends comparing our contribution with other high-performing HTM systems and adding few concluding remarks.

6.1 Motivation

Nowadays, most HTM systems implement fixed (either eager or lazy) version and conflict management mechanisms. Unfortunately, fixed-policy HTM systems are faced with several challenges that limit the concurrency of transactional workloads [16].

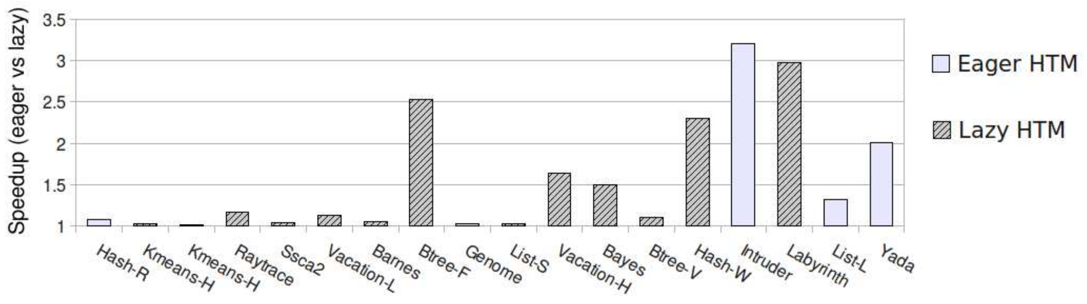


Figure 6.1: Speedup over opposite fixed-policy (eager or lazy) HTM systems

Inflexible conflict management has to prioritize between conflicting transactions. On the one hand, lazy HTM systems must abort all the transactions that conflict with the committing one, which (i) may result to starvation of the older transactions [14] and (ii) it increases the amount of discarded transactional computation [111, 122]. On the other hand, eager HTM systems may abort a transaction multiple times, which may lead to different pathological situations [98]. Nevertheless, lazy transactions can avoid some read-write conflicts whereas eager transactions minimize discarded work in the case of write-write violations by stalling conflicting requesters [14].

As we have mentioned in Section 5.4.4, there is no fixed-policy HTM system that outperforms the rest. Figure 6.1 shows, for transactional benchmarks, which high-performing eager (FASTM) or lazy (TCC-Loc) HTM system obtains better performance—the height of each bar shows the performance improvement over the other HTM system. As it can be seen, applications with some contention and small transactions (*e.g.*, *Raytrace* or *Vacation-high*) or those with read-write conflicts (*e.g.*, *Btree-fix* or *Hash-write*) yield better performance when they are executed in lazy HTM systems, while applications with high contention and large transactions (*e.g.*, *List-long* or *Yada*) obtain better results when they are executed in eager HTM systems. In some applications, the performance gap between eager and lazy HTM systems is huge—close to 3X speedup in *Labyrinth* (lazy better) or *Intruder* (eager better).

Figure 6.2 shows the benefits of using flexible conflict management. In Situation 1, eager transactions stall their execution when they find a read-write conflict, while lazy transactions speculate with the conflict and commit without aborting if the reader finishes before the writer.

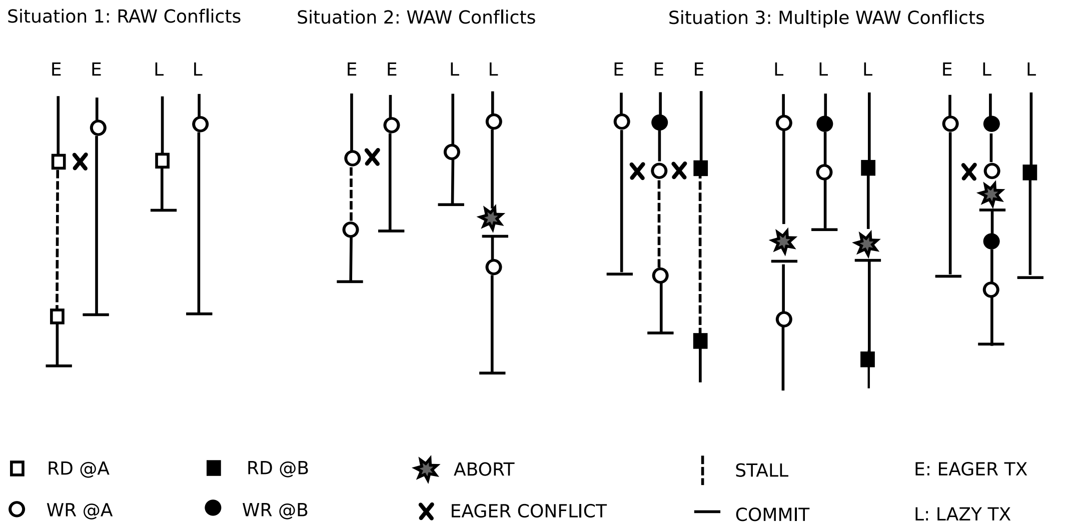


Figure 6.2: Conflict management in eager, lazy and dynamically adaptable HTM systems

In Situation 2, however, eager transactions do not abort when they discover a write-write conflict, and thus they conserve transactional computation. Having a flexible VM and CM scheme should allow the system to select the most profitable policy on each situation.

In addition to this, complex applications that combine small and large transactions with variable contention present a great challenge for HTM systems that fix the version and conflict management strategies for the whole program execution: while eager HTM systems can preserve the computation generated by long transactions in case of collision, lazy HTM systems are more effective dealing with small, high-contending transactions. A truly flexible HTM that could select the ideal (eager or lazy) execution mode for each transaction at runtime would not be challenged by such complex design choice. In Situation 3 of Figure 6.2, we present how three different transactions interact in an HTM system that can choose at free will the transactional mode of execution. When transactions are executed under the *same* mode (analogous behavior to a fixed-policy HTM system), they perform poorly. However, when transactions are executed using *distinct* modes, overall throughput is improved.

On top of that, some applications present a level of contention that fluctuates through time. This dynamic behavior can be found in workloads with transactions that operate on data structures that continuously modify their size (*e.g.*, binary trees). At the beginning, the tree is empty, so each insertion provokes a conflict—threads simultaneously attempt to add a leaf that is linked with the root. However, the number of collisions diminishes as the tree increases its size, given that insertions are spread over distinct branches of the tree. Inflexible strategies are not efficient on those situations, because all the instances of the transaction use the same policies. An HTM that adapts its version and conflict management mechanisms should be able to catch the dynamic behavior of these kind of applications.

6.2 A Dynamically Adaptable HTM System

DynTM (dynamically adaptable HTM system) is the first HTM system that *cleverly* combines eager and lazy transactions to untie the conflict management policy from the bottom-line HTM machinery. Using FuseTM's infrastructure, DynTM describes a runtime prediction scheme that decides, for each *dynamic* instance of a transaction, at what mode it should be executed according to its characteristics. When tightly coupled with a new conflict resolution policy, this system enables safe and efficient execution of eager and lazy transactions.

6.2.1 DynTM Overview

DynTM offers two different execution modes (eager and lazy) that use opposite VM and CM strategies. DynTM incorporates the UTCP protocol to isolate eager modifications from other in-flight transactions and to track violations between lazy conflicting accesses. Like FuseTM, DynTM takes advantage from the deep-seated early VM support in order to handle cache overflows and context switches for large transactions.

Contrary to FuseTM, in DynTM the system *dynamically* decides the most profitable (or the necessary) execution mode at the beggining of a transaction. That choice is preserved until commit or abort time. DynTM selects the eager or lazy mode of execution by consulting a per-core Transactional Mode Selector (TMS). This hardware component uses past information of the *current* instance of the transaction (if the transaction aborts there will be several retries of the same instance) and the history of *previous* instances of the same transaction to determine the most effective execution mode.

Predicting the behavior at runtime permits the system to select the best-suited policy for each *individual* instance of a transaction. This scheme allows DynTM to break the chains imposed by fixed-policy HTM systems, which lack adaptability.

6.2.2 Programming Model

Like most HTM systems, DynTM only needs two new instructions to define the boundaries of the transactions: `TM_BEGIN()` and `TM_END()`. All the memory accesses performed inside the atomic block are treated as transactional, requiring special operations. Although DynTM does not require additional instructions, we have added two new directives that permit the programmer to take control over the VM and CM mechanisms.

We have introduced the `TM_CONFLICT(mode)` directive, where mode can be `EAGER` or `LAZY`, that forces all transactions of the application from this point on to run at the execution mode selected by the user. This execution modes are analogous to the ones presented in FuseTM. Hence, if a transaction executed after setting a `LAZY` environment overflows, it has to abort and restart in eager mode, just as in FuseTM. In contrast, if the transaction is executed after setting a `EAGER` environment, it will run in FuseTM's eager mode from the very begginnig.

We also add the `TM_BEGIN(mode)` directive, which statically indicates the execution mode for all the instances of the defined transaction. This second directive has higher precedence over `TM_CONFLICT`. These two directives allow expert programmers to override the default execution mode selected by the system and combine eager and lazy transactions on the same application.

6.2.3 Transactional Mode Selector

In DynTM, each core includes a simple Transactional Mode Selector (TMS) to decide the most profitable execution mode for each *proper* transaction. The appropriate execution mode for a transaction is highly application-dependent. Lazy transactions usually manage contention more efficiently than eager transactions, especially when there are many small transactions with high contention. Nonetheless, eager transactions reduce the amount of discarded work due to aborts of large transactions. For this reason, the TMS decides to execute most of the transactions lazily, except in the case of multiple lazy-mode aborts or frequent overflows.

The TMS configuration shares similarities with typical two-level branch predictors [129]. As it can be seen in Figure 6.3, the TMS requires two hardware structures that store important information about past transactional executions. The first structure is the Transactional State Register (TSR), which collects information about the current *dynamic* instance of a *locally*

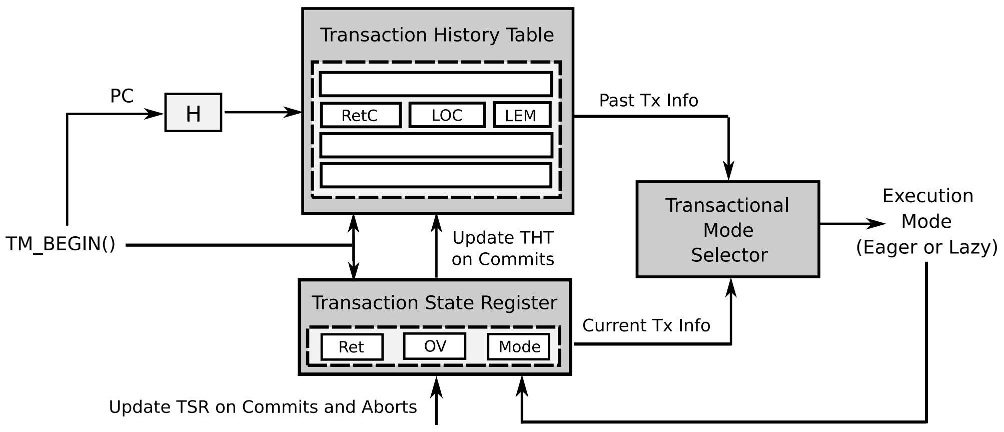

Transaction History Table
PC
H
RetC
LOC
LEM
Past Tx Info
Transactional Mode Selector
Execution Mode (Eager or Lazy)
TM_BEGIN()
Update THT on Commits
Transaction State Register
Ret
OV
Mode
Current Tx Info
Update TSR on Commits and Aborts

Figure 6.3: Hardware support for the Transactional Mode Selector

executing transaction. The second structure is the Transactional History Table (THT), which records statistics from previously committed transactions on this core.

The TSR contains (i) the overflow bit (OV), which is asserted when the system aborts a lazy transaction due to an L1 cache overflow, (ii) a 3-bit saturating counter (Ret) that counts how many aborts (*i.e.*, retries) the currently executing transaction has performed, and (iii) the Mode bit, which determines the execution mode of the current in-flight transaction. Each entry of the THT has two 2-bit saturating counters and a bit that contains the execution mode of the last committed instance of the transaction (LEM bit). The first counter (LOC) tracks if the transaction is prone to overflow while the latter (RetC) tracks if the transaction is prone to abort multiple times before committing.

At the beginning of a given transaction, the TMS decides the execution mode (eager or lazy) of the transaction and stores the decision in the Mode bit of the TSR. This decision is preserved until the transaction commits or aborts. Figure 6.4a shows how the execution mode is selected using the TMS. The TMS uses the TSR when the system re-executes an aborted transaction (Ret>0). In this case, DynTM changes the execution mode from lazy to eager when (a) the OV bit is asserted or (b) the number of transactional retries is above a threshold *T*. In our evaluation, the threshold *T* is a static parameter (the number of active threads divided by four). This technique permits our system to eliminate the *starvation of the older* pathology [14] and minimize the amount of discarded transactional computation [111].

(a) Execution Mode Predictor

```
if(Ret>0)
   if(OV == true || Ret > T || Mode == Eager)
      Mode = Eager
   else
      Mode = Lazy
else
   if(LOC == 3 || RetC == 3)
      Mode = Eager
   else if (LOC < 2 && RetC < 2)
      Mode = Lazy
   else
      Mode = LEM
```

(b) Transactional History Table Update

```
if(Ret > 2*T && RetC < 3){
      RetC++
else if(Ret < T/2 && RetC > 0)
      RetC--

if(OV == true && LOC < 3){
      LOC++
else if(OV == false && LOC > 0)
      LOC--

LEM = Mode
```

Figure 6.4: TMS selection (top) and THT update (bottom) algorithms

When a new instance of a transaction starts (*i.e.*, not a re-execution), the TMS indexes the THT with the Program Counter (PC) of the transaction to decide the execution mode. Like in conventional Branch Predictors, the PC goes through a hash function to avoid aliasing [129]. If it hits in the THT, the TMS inspects the corresponding saturated counters. If previous instances of the same transaction have presented a recognizable behavior (confident LOC or RetC counters), the TMS chooses between the eager (high counter values) or lazy (low counter values) execution modes.

If the predictor is not confident on its decision, the TMS chooses the execution mode used in the last committed instance of the transaction (LEM bit). If there is a miss in the THT, the TMS executes the transaction lazily because lazy transactions usually obtain better performance than

	Lazy Reader	Lazy Writer
Eager Reader	No conflict	Speculate with the eager reader Abort eager transactions if the lazy transaction commits first
Eager Writer	Abort lazy transaction (immediately)	Abort lazy transaction (immediately)

Table 6.1: Resolving eager-lazy conflicts in DYNTM

eager transactions. The THT is updated each time the core commits an instance of a transaction following the algorithm described in Figure 6.4b.

Notice that all the operations that involve the TMS—TSR/THT lookups and updates—are performed *locally* using information from transactions executed in the same core. Hence, the TMS does not suffer scalability issues.

6.2.4 A Highly-Efficient Policy for Eager and Lazy Transactions

DYNTM introduces a novel conflict resolution policy that enforces the right outcome for solving conflicts between eager and lazy transactions. Like in FUSETM, this policy schedules eager transactions over lazy transactions, although is less restrictive than the prior policy, as it permits to speculate with some read-against-write conflicts.

In DYNTM, lazy transactions cannot safely access the pre-transactional data of an eager transaction because, in the case of a transactional L1 cache eviction, eager transactions write back the line in the L2 cache, polluting the pre-transactional values. For this reason, lazy transactions must abort when they access a memory location written by an eager transaction, since they cannot know if the L2 cache contains a pre-transactional or an evicted eager value.

Nonetheless, eager readers speculate when they conflict with lazy writers. When an eager transaction wants to read data that is written in a lazy transaction, the system will respond with the line from the L2 cache (lazy modifications are never evicted from the L1 cache, so the L2 cache always keeps the pre-transactional state) and mark a conflict in the eager transaction's RCV vector. This policy avoids read-write and write-read conflicts if the eager transaction commits before the lazy transaction. If the lazy transaction commits first, then the eager transaction must abort. Notice that eager transactions only speculate with read data (the WCV remains

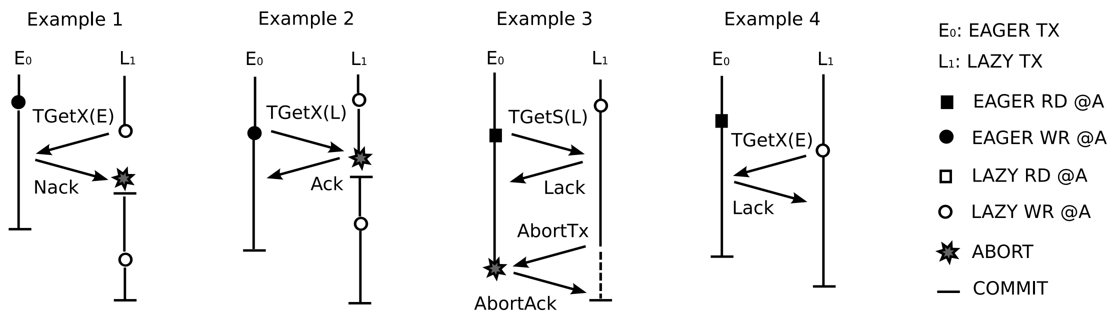


Figure 6.5: Resolving eager/lazy conflicts in DYNTM

empty), so abort notification at commit time is not required. Table 6.1 summarizes the conflict resolution policy between eager and lazy transactions.

Now, assume that E_0 is a core executing an eager transaction and L_1 is a different core executing a lazy transaction. In order to explain how conflicts that involve eager and lazy transaction are resolved, we will describe the various situations in Figure 6.5.

Eager Early Write (Example 1): DYNTM must prevent lazy transactions from reading or writing the modifications introduced by eager transactions. Thus, when L_1 attempts to access a line being modified by E_0, E_0 responds with a *Nack* message. After receiving the *Nack* response, L_1 aborts immediately.

Eager Late Write (Example 2): Similarly, upon a write request from E_0, L_1 acknowledges the request and aborts itself, permitting the eager transaction to obtain the pre-transactional data from the L2 cache. This is safe to do because lazy writes never leave the L1 cache. This approach reduces the amount of wasted computation on aborts and facilitates fast restarts, since lazy transactions do not require backoff cycles.

Eager Late Read (Example 3): When E_0 reads data that is written in L_1, L_1 responds with a *Lack* message. E_0 marks the conflict in its RCV, and L_1 marks the conflict in its WCV. E_0 receives the line data from the L2 cache and stores it in the R state. Since lazy modifications are never evicted from the L1 cache, E_0 gets the correct pre-transactional data. This policy avoids aborts from read-write conflicts when E_0 commits before L_1. Of course, if L_1 commits first, E_0 has to abort.

Eager Early Read (Example 4): Similarly, L_1 can continue its execution when it writes a memory location that has been read by E_0, tracking the conflict in its WCV. Hence, if L_1 com-

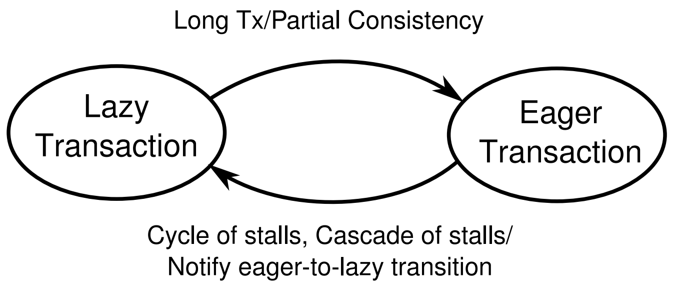

Long Tx/Partial Consistency
Lazy Transaction
Eager Transaction
Cycle of stalls, Cascade of stalls/ Notify eager-to-lazy transition

Figure 6.6: Transiting from eager to lazy and vice versa

mits before E_0, an *AbortTx* message is sent to E_0, which immediately aborts. Otherwise, if E_0 commits before L_1, no conflict is reported.

6.3 A High-Performing HTM with Swapping Execution Modes

DynTM offers a thorough solution to break with the inflexibility of HTM systems. However, it still imposes two major concerns that may affect the performance of TM applications with *irregular* transactions. First, it forces early VM for those transactions that exceed the limits of the L1 cache. This becomes a problem if the TMS fails in its prediction and decides to execute the transaction in lazy mode because then overflowing transactions have to be aborted. Second, DynTM assigns a mode of execution when transactions start, and that decision is preserved until commit or abort time. This may lead to pathological situations that should be avoided if the transaction could *switch* its mode of execution on demand.

6.3.1 SwapTM Overview

SwapTM (high-performing HTM with swapping execution modes) is the first HTM system that can adapt on the fly the most profitable mode of execution without causing unnecessary aborts. The system relies on simple hardware to collect information of the current instance of a transaction and then uses this knowledge to dynamically change the transactional execution mode. Of course, switching from a mode to another may trigger additional actions to satisfy the constraints of each mode of execution.

For SwapTM we assume a similar hardware foundation to DynTM, which allows the system to execute eager and lazy transactions simultaneously. However, there is a key difference between both systems: like in SpecTM, this system incorporates selective logging (and the rest of mechanisms) to implement early VM for overflowing data. In such way, SwapTM is able to decouple the conflict management technique from the size of a transaction.

SwapTM decides to move to the eager mode those transactions that have been running for a long time. Experiments carried out in this thesis show that it is critical to prioritize large transactions because otherwise they may starve by younger transactions (this happens when all transactions are executed in lazy mode [14]). We consider a transaction large when the number of transactional reads or writes that it performs is considerable higher than the average.

SwapTM may also switch from eager to lazy at runtime. There are two situations to perform such transition. First, when an eager transaction continuously clashes with other transactions may produce a *cascade of stalls* (a transaction that is waiting for a conflict to be resolved owns data requested by a third transaction) or a *contention point* (a single transaction is continuously denying access to a group of transactions that also have high-priority). Moving transactions that generate contention to a more relaxed conflict management strategy increases the transactional activity in the system (higher commit rates) and enables speculation on read-write conflicts (lower abort/stall rates).

Second, if there is a cycle between stalled transactions, conventional eager HTM systems must abort at least one transaction. Someone may claim that these aborts are needed anyway, because transactions involved in a cycle are not serializible. This is not entirely true, because moving conflicting transactions to lazy mode favors overall concurrency (some of the read-write conflicts may disappear) and guarantees forward progress (in eager mode, the system must resolve cyclic dependences taking *blind* decisions, which may introduce repeated executions of transactions if the winner of the conflict is aborted by a third transaction).

To prevent contention points and repeated aborts, the system records the number of conflicts generated by eager transactions. Figure 6.6 shows SwapTM mode transitions and some of their associated actions. Conflicts between transactions are resolved using the same resolution policies to DynTM, which are briefly described in Table 6.1

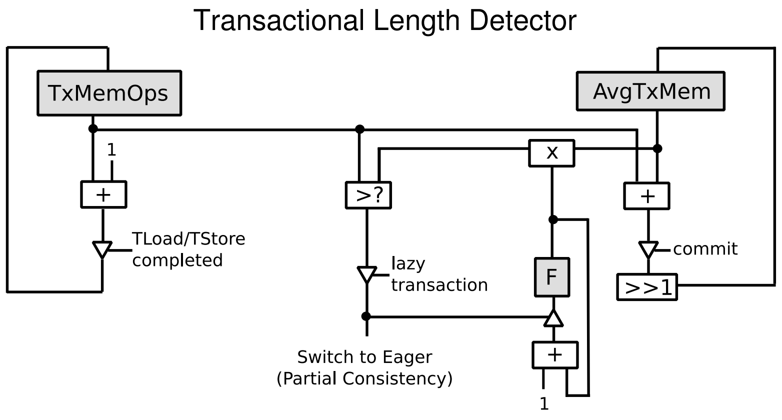


Figure 6.7: Detecting long transactions in SWAPTM

6.3.2 Hardware Support

Besides DYNTM's conventional hardware (UTCP protocol, conflict vectors, *etc.*) and selective logging support, SWAPTM incorporates specialized structures to keep account of transaction's properties.

Long Transaction Detector (LTD). In SWAPTM, each core tracks in a local register (called `TxMemOps`) the number of transactional loads and stores that are completed successfully. Thus, when a transaction starts the register is cleaned, and then incremented each time that a store is retired inside this transaction. Cores also keep in another register (called `AvgTxMem`) an approximation of the average number of stores performed by the already committed transactions in the core. This is a pondered average, which is calculated each time an instance of any transaction ends using the following formula:

$$\texttt{AvgTxMem}_i\texttt{:=(AvgTxMem}_{i-1}\texttt{+TxMemOps)/2}$$

Basically, the new average becomes the mean between the old average and the number of memory operations of the committing transaction. Hence, recent committed transactions have more weight than the older ones. Cores detect large transactions when the number of memory operations completed in the actual transactional instance is higher than the average (multiplied by a factor *F* to prevent cyclic transitions from lazy to eager, more on this later). Figure 6.7 shows a scheme of the LTD mechanism.

Eager Starvation Tracker (EST). In SWAPTM, each core records in its EST how many transactions it has stalled. When this counter bypasses a threshold *T*, the system assumes that the in-flight transaction is starving other eager transactions.

SWAPTM uses the underlying hardware to determine if the current transaction must switch its mode of execution. In some occasions, these transitions carry additional actions. In the next section, we explain when transactions change their mode of execution and which mechanisms they trigger.

6.3.3 SWAPTM Execution Mode Transitions

SWAPTM uses the underlying hardware to determine if the current transaction must switch its mode of execution. We will use the examples from Figure 6.8, which recreate distinct situations where transactions dynamically shift their policies, to describe how system transits from/to distinct execution modes. Eager transactions are represented with a continuous line, while lazy transactions are represented with a dotted line.

Moving from lazy to eager (long transaction): In Example 1 of Figure 6.8, transactions `Ti` and `Tj` are being executed in lazy mode. At some point in time, SWAPTM determines that `Tj` transaction must be executed in eager mode, increasing its priority level over `Ti`. This happens when the LTD mechanism detects that the number of memory accesses performed within a transaction is over a threshold (step 1). This threshold is calculated using the pseudo-average `AvgTxMem` multiplied by a factor *F*, which original value is one.

Eager transaction do not accept inconsistencies, as they only permit a unique owner per transactionally written line. Thus, when `Tj` reaches the maximum number of memory operations, it stops normal execution and activates the *Partial Consistency* machinery, which notifies the abort of conflicting transactions (in this case, `Ti`), who clear inconsistent data from their caches (step 2). Like in SPECTM, this mechanism inspects the WCV and sends point-to-point abort messages to all conflicting transactions. Note that this is the same mechanism that SWAPTM uses each time a core evicts a `W`-state cache line—a transactionally written line that has been accessed in other non-committed transactions.

Once the *Partial Consistency* action finishes, `Tj` switches to eager mode. To eliminate repetitive lazy-to-eager transitions, each time that a transaction moves from lazy to eager the factor *F* is

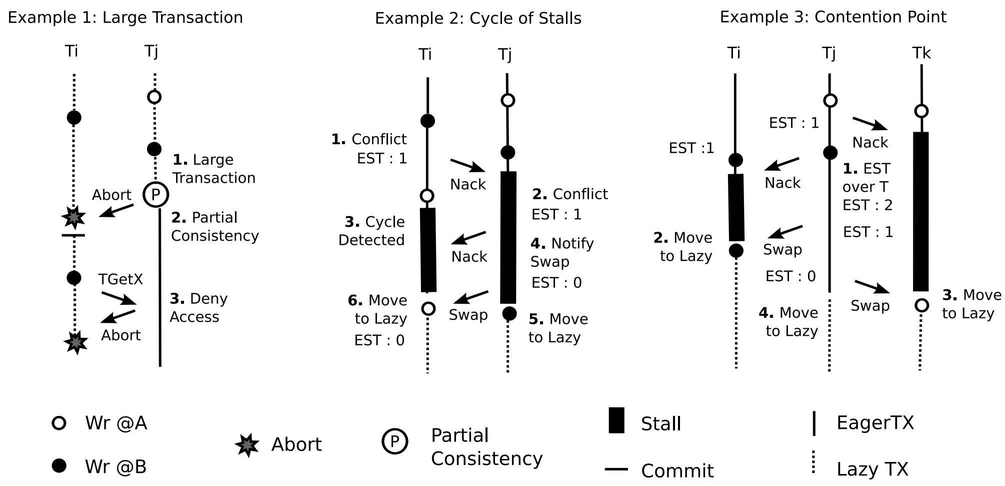


Figure 6.8: Switching execution modes in SWAPTM

incremented. Hence, if `Tj` transits to lazy again in a near future, it would not change automatically to eager—`Tj` will only do that when it doubles the current size.

Eager transactions must prevent remote transactions to access their write set. In SWAPTM, eager transactions have higher priority than lazy transactions, so when `Ti` asks for data owned by `Tj`, `Tj` replies with an abort message (step 3). This guarantees that the longer transaction that is being executed in eager mode (in this case, `Tj`) will commit in a near point in time.

Moving from eager to lazy (cycle between stalled transactions): In Example 2 of Figure 6.8, `Ti` has written data *B* and `Tj` has written data *A*. As both transactions are eager transactions, they must prevent requesting transactions to access their read/write set (by sending *Nack* notifications and updating their EST counters). After receiving a *Nack* message, they must retry the non-completed memory access. This scenario happens when `Tj` attempts to obtain the ownership of *B* (step 1) and when `Ti` attempts to get access to *A* (step 2). Hence, transactions `Ti` and `Tj` have crossed conflicts.

SWAPTM is able to identify these cycles by adding timestamps on transactional accesses [96]. When this occurs, the youngest transaction that participates in the cycle (in this example, `Tj`) transits to lazy mode to eschew an abort (step 3). The transition to lazy mode is not immediate, as `Tj` has to be sure that all conflicting transactions transit to lazy mode as well. Otherwise `Tj` would see a conflict with an eager transaction once it retries the conflicting access, and it would automatically abort.

Hence, before jumping to lazy mode, `Tj` must inform to all its stalled transactions (in this case, `Ti`) that they must also switch to lazy execution. This is done by replying each conflicting request with a *Swap* message, and decrementing the EST counter (step 4). Only when the EST counter is zero `Tj` can resume its execution in lazy mode—this means that, in the normal case, all conflicting eager transaction have already been informed that they should transit to lazy mode (step 5). After receiving a *Swap* reply, `Ti` moves to lazy mode and resends the offending request (step 6).

Moving from eager to lazy (contention point): In Example 3 of Figure 6.8, three transactions `Ti`, `Tj` and `Tk` are executed in eager mode. At a particular point, transaction `Tj` starves `Ti` and `Tk`, placing the EST counter above a threshold *T* (step 1). (In SWAPTM, *T* is the number of active threads divided by four, although to ease the comprehension of this example we have set it to two).

When the threshold is reached, SWAPTM decides to move transaction `Tj` and all its *nackers* to lazy mode, which increases the concurrency of parallel threads. Hence, when `Ti` and `Tk` retry their conflicting memory access, `Tj` replies with *Swap* notifications, so they also can transit to lazy mode (step 2 and 3). Only when all stalled transactions are revived in lazy mode (EST becomes zero) `Tj` performs the switch to lazy mode (step 4).

Note that there can be the case when a new requester conflicts with the transaction that is carrying the eager-to-lazy transition. This is not problematic in terms of correctness, given that the new requester would transit to lazy (if it is eager) or retry the access (if it is already lazy). However, it may introduce a performance pathology, given that an eager transaction (the one that does not receive the *Swap* notification) will not transit to lazy, becoming the only participant of the conflicting group with high privileges. In the worse scenario, this will generate an abort of the transaction that produced the contention point (now in lazy mode) when it clashes against the non-transiting one (still in eager mode).

6.4 Evaluation

For our evaluation of DYNTM, we have equiped each core with 2Kbit signatures, 32-bit Conflict Vectors (one bit per core) and a TMS with a 16-entry THT. A 16-entry predictor is enough to avoid aliasing between different transactions. We also evaluate three distinct

DynTM alternatives. **DynTM-Ov** uses a simpler predictor that only reports if the transaction is prone to overflow, whereas **DynTM-Ab** predictor only indicates when the transaction is prone to abort a lazy transaction multiple times. The original DynTM uses the policy introduced in this chapter to resolve eager-lazy conflicts, while **DynTM-EP** implements the *eager priority* policy by FuseTM.

For our implementation of SwapTM, we assume analogous hardware support to DynTM except for the TMS. To quantify the importance of switching modes at any point of time, we evaluate two additional systems: **SwapTM-TLD** only performs a single transition from lazy to eager mode when the number of transactional writes is beyond the average, whereas **SwapTM-EST** adds eager-to-lazy transitions when a cascade of stalls is detected (all SwapTM features but eager-to-lazy transition after detecting a cycle of stalled transactions).

We compare DynTM (labeled D in all the figures) and SwapTM (labeled W in all the figures) against the following fixed and dynamic HTM systems:

FasTM-IVM (labeled E): This configuration is our eager HTM baseline, which corresponds to a FasTM implementation that uses the idealized VM system proposed in Chapter 4 to spend zero cycles in aborts and commits. Conflicts are *always* resolved eagerly using the **Hybrid** conflict resolution policy. Thus, this HTM system can be seen as an upperbound of FasTM.

TCC-Loc (labeled L): This is our lazy HTM baseline, which corresponds to the HTM system introduced in Chapter 5. It uses an idealized data versioning for overflowing data and local commits with core-to-core abort notification. Conflicts are *always* resolved lazily at commit time, borrowing the strategy from TCC [45] or EazyHTM [124].

FuseTM (labeled F): This HTM system executes all transactions in lazy-mode except those that exceed the L1 cache. We evaluate FuseTM with its original policy (eager priority) and with the high-performance policy introduced in this chapter—we named that system **FuseTM-HP**.

SpecTM (labeled S): This HTM system executes all transactions in lazy-mode, but resolves conflicts when a transaction evicts speculative data. Moreover, the system prevents the access of those replaced blocks by aborting requesters on the fly.

Statically Programmed HTM (labeled P): Static alternative to DynTM where an expert programmer decides the execution mode of transactions using the `TM_BEGIN(mode)` directive. For our evaluation, we decided to execute all transactions lazily except those transactions that overflow the L1 cache or those transactions with many lazy aborts. We use this system to evaluate the performance benefit of the TMS predictor compared to a simpler adaptive method.

6.4.1 DYNTM Performance Analysis

Figure 6.9 presents the time distribution of eager FASTM (ideal VM, labeled E), lazy TCC-Loc (ideal VM, labeled L) and DYNTM (non-ideal VM, labeled D) in their 32-threaded executions (for low-contention applications, top of Figure 6.9) and in their 16-threaded executions (for high-contention, bottom of Figure 6.9) applications. The execution time has been normalized to the 32-threaded (low-contention) and 16-threaded (high-contention) FASTM execution and is broken down to: non-transactional and barrier cycles (labeled Non-Tx and Barrier), the time spent in committed transactions (labeled Good Tx), the time that is wasted in non-useful work discarded from aborted transactions (labeled Aborted Tx), the time spent in abort recovery and in the commit phase (labeled Aborting and Commit), the time that transactions remain stalled waiting for a conflict to be resolved (labeled Stalled), and the time that processors execute the exponential backoff after aborting (labeled Backoff).

DYNTM outperforms both fixed policy systems by (i) combining eager and lazy transactions in applications that execute heterogeneous transactions and (ii) re-adapting the execution mode of the transactions at runtime. As it can be seen in Figure 6.9, DYNTM achieves, on average, a speedup of 19% over FASTM-IVM and a speedup of 57% over TCC-Loc in high-contention applications. In low-contention applications, the speedup is not so impressive, but significant over FASTM-IVM (13% improvement).

Similarly, Figure 6.10 shows the time distribution of DYNTM (labeled D) together with two realistic HTM systems that use either fixed or static conflict management strategies: FUSETM (labeled F) and the Statically Programmed version of DYNTM (labeled P). As it shown, DYNTM obtains an average speedup of 24% (6%) on high-contention (low-contention) applications over FUSETM and a 12% (7%) with respect to the Statically Programmed HTM system. The reasons why DYNTM outperforms state-of-the-art fixed and static HTM executions are described in the following paragraphs.

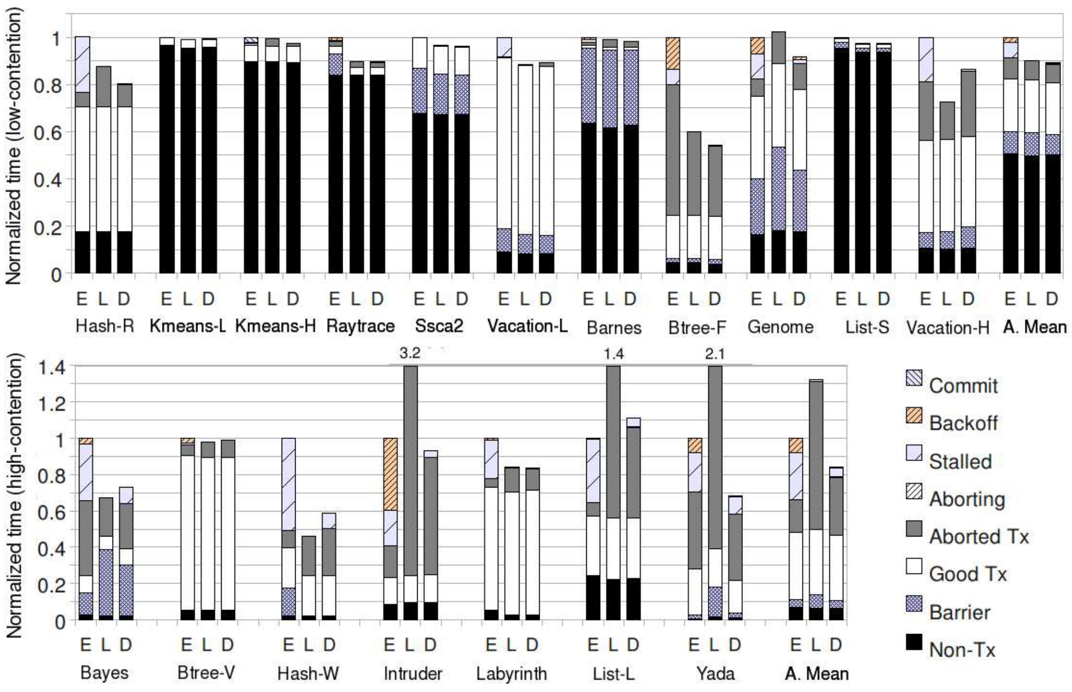


Figure 6.9: Distributed executed time of low- and medium-contention (top, 32 threads) high-contention (bottom, 16 threads) TM applications under FASTM-IVM (E), TCC-Loc (L) and DYNTM (D) HTM systems

Truly flexible system. DYNTM uses the eager execution mode for transactions that (i) commonly overflow the L1 cache and (ii) for transactions that abort several times before committing. In the former case, DYNTM can stall large transactions—those that modify a lot of lines—to preserve useful work in case of conflict without requiring specialized late VM support (*i.e.*, the transactional victim cache used in TCC-Loc, which speeds up the execution of *Vacation-high*). In the latter case, older transactions can commit faster, decreasing the number of aborts—eager transactions have more priority than lazy transactions in DYNTM. Figure 6.9 shows that combining eager and lazy execution has a positive effect in applications with heterogeneous transactions like *Genome*, *Bayes*, *Intruder* or *Yada*, which reduce the Stalled and Backoff cycles (with respect to FASTM) and the Aborted Tx cycles (with respect to TCC-Loc).

Catch dynamic behavior. Figure 6.10 shows the importance of having a dynamic execution mode selector. FUSETM cannot combine execution modes on applications like *Intruder*, which starve older non-overflowed transactions. Moreover, overflowed transactions must abort before

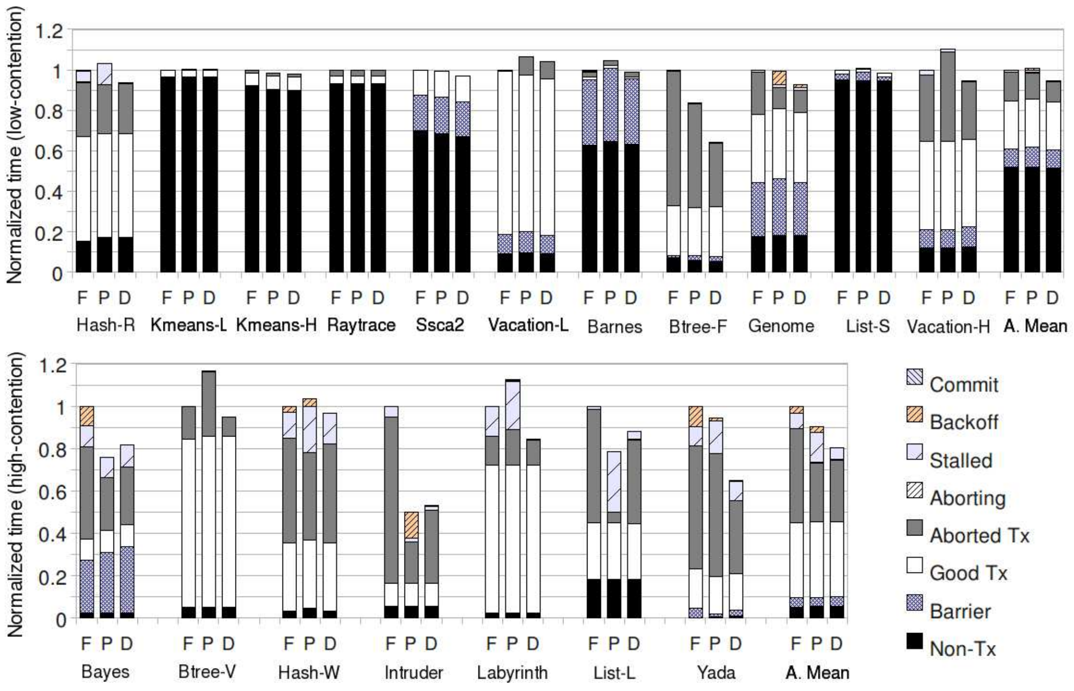


Figure 6.10: Distributed executed time of low- and medium-contention (top, 32 threads) high-contention (bottom, 16 threads) TM applications under FUSETM (F), Statically Programmed (P) and DYNTM (D) HTM systems

being re-executed, which increases the amount of discarded transactional work in *Yada*. In contrast to FUSETM, DYNTM does not need to abort lazy overflowed transactions and then restart them in eager mode. It recognizes very quickly which transactions will probably overflow, and decides to executes most of them eagerly right away.

The Statically Programmed HTM delegates the election of the execution mode to the programmer. The assignment that the programmer has performed tries to minimize the impact of aborts caused by overflows (*e.g.*, in *Bayes*) and accelerate applications with multiple lazy aborts (*e.g.*, in *Intruder*). However, applications that present a dynamic behavior (such as phase changes) may suffer considerable delays when we fix the execution mode of a transaction for the entire application. This happens in applications like *Genome*, *Btree-var* or *Yada*, which present several overflows at the beginning of the execution and less overflows at their end.

As opposed to the Statically Programmed HTM, DYNTM executes eager transactions only when it is necessary (when lazy aborts are frequent), avoiding the shortcomings of using conservative

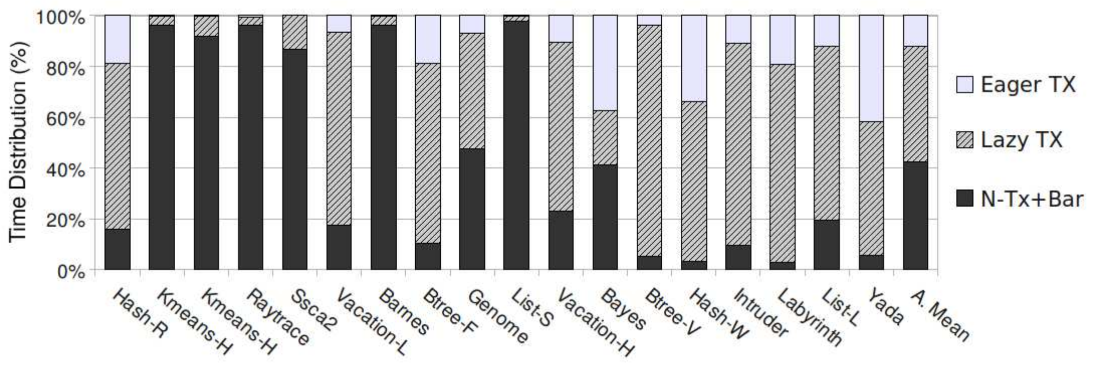


Figure 6.11: Normalized DYNTM execution time of applications distributed by the transactional mode

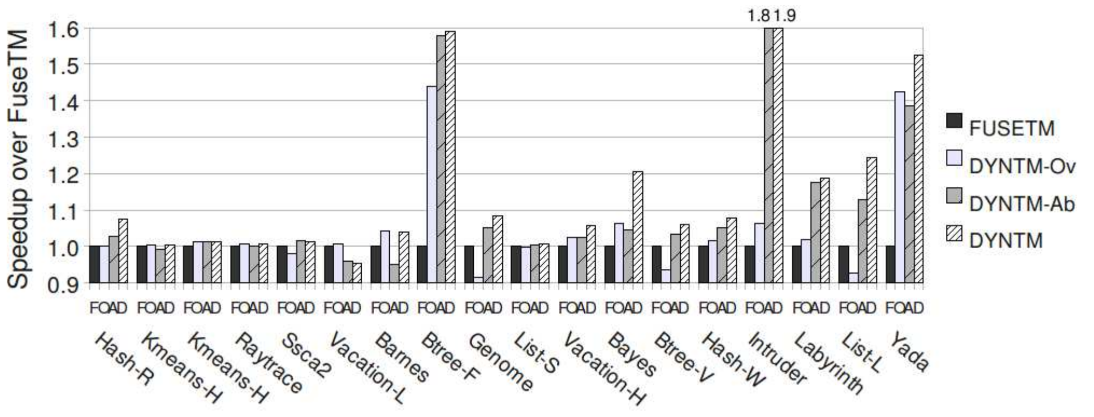


Figure 6.12: Speedup achieved in low-contention (32-threads, left) and high-contention (16-threads, right) applications by FUSETM, DYNTM-Ov, DYNTM-Ab and DYNTM

conflict management mechanisms for the entire application. The only scenario where the Statically Programmed HTM system performs better than DynTM is *Bayes* and *Hash-write*. This happens because *Bayes* only executes few transactions, which does not give enough time to our dynamic selector to learn the best execution mode for each transaction.

Best-suited execution mode selection. By re-adapting the system at runtime, DYNTM can use the most profitable strategy through the whole execution. Figure 6.11 breaks down applications run on top of DYNTM according to the mode of execution. As it can be seen in the figure, DYNTM executes most of the applications with small transactions lazily. This strategy is really useful because it (i) eliminates read-write conflicts if the reader commits before the writer, (ii)

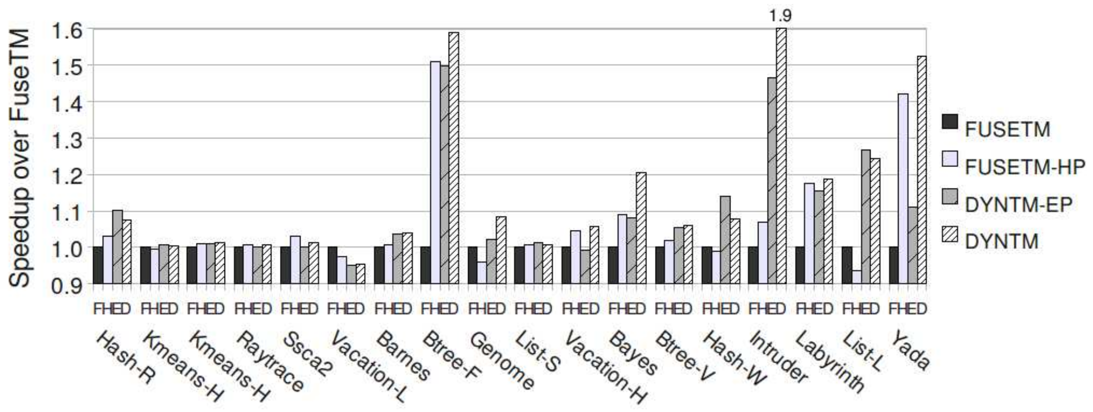


Figure 6.13: Speedup achieved in low-contention (32-threads, left) and high-contention (16-threads, right) applications by FUSETM, FUSETM-HP, DYNTM-EP and DYNTM

does not require exponential backoff and (iii) removes pathological behavior caused by stalled transactions. In contrast, coarse-grain applications spent more time in the eager mode, given that large transactions that overflow the L1 cache do not support the lazy execution mode and conservative conflict management reduce drastically the number of aborts and re-executions.

High-accurate predictor. Selecting the best-suited execution mode for each individual task may become a delicate task. Nonetheless, DYNTM's TMS does a great job handling this assignment. Figure 6.12 shows the speedup achieved by DYNTM-Ov, DYNTM-Ab and DYNTM with respect to the FUSETM execution. As it can be seen, moving overflowing transactions from eager at the beginning enhances the performance of *Btree-fix* or *Yada*. However, in applications with read-only conflicts like *Genome* having only an overflow predictor can have a negative effect. In other applications like *Intruder* its necessary to count the number of lazy aborts to increase the priority of critical transactions. When both techniques are combined, the predictor finds the most profitable mode of execution in most of the occasions. *Vacation-low* is the only benchmark that performs worse in DYNTM over FUSETM. The reason of such behavior lies on the erratic style of its transactions, which makes difficult for the predictor to guess which will be the adequate mode.

High-performing conflict policy. Part of the performance improvement that DYNTM yields comes from the policy employed to resolve eager-lazy conflicts. Figure 6.13 shows how both FUSETM and DYNTM perform under the *eager win* policy (original FUSETM and DYNTM-EP)

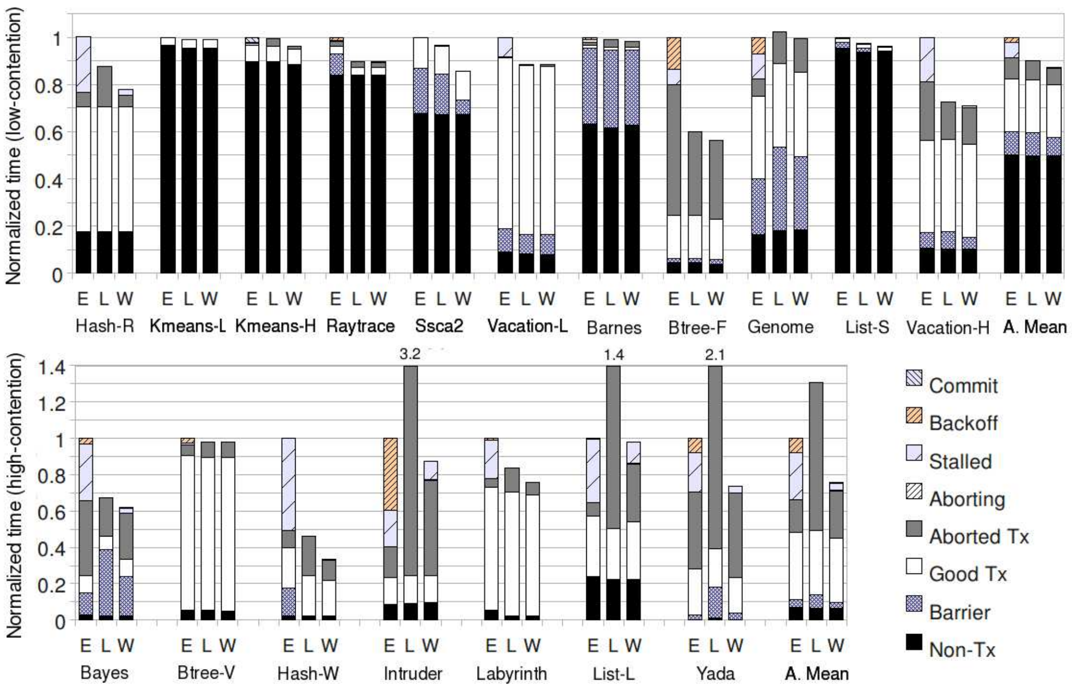


Figure 6.14: Distributed executed time of low- and medium-contention (top, 32 threads) and high-contention (bottom, 16 threads) TM applications under FASTM-IVM (E), TCC-Loc (L) and SWAPTM (W) HTM systems

and the *high-performance* policy (FUSETM-HP and the original DYNTM). As it can be seen, FUSETM can take advantage of the new policy in applications like *Btree-fix*, *Labyrinth* or *Yada* to increase overall concurrency. However, some applications with crossed conflicts (*e.g.*, *Hash-read* or *List-long*) perform better (both for FUSETM and DYNTM) with the *eager wins* policy.

6.4.2 SWAPTM Performance Analysis

Figure 6.14 exhibits the performance improvement of SWAPTM (labeled W) compared with FASTM-IVM (labeled E) and TCC-Loc (labeled L). As it can be observed, SWAPTM achieves, in low-contention benchmarks, a 15% speedup over the ideal FASTM implementation and a 4% speedup over TCC-Loc. The benefits of SWAPTM are especially notable in workloads that require dynamic conflict management for variable-size transactions (*e.g.*, *Hash-read* and *Vacation-high*). In high-contention applications, the performance gap between FASTM-IVM (32%), TCC-Loc (76%) and SWAPTM is bigger due to the complex nature inherent of those

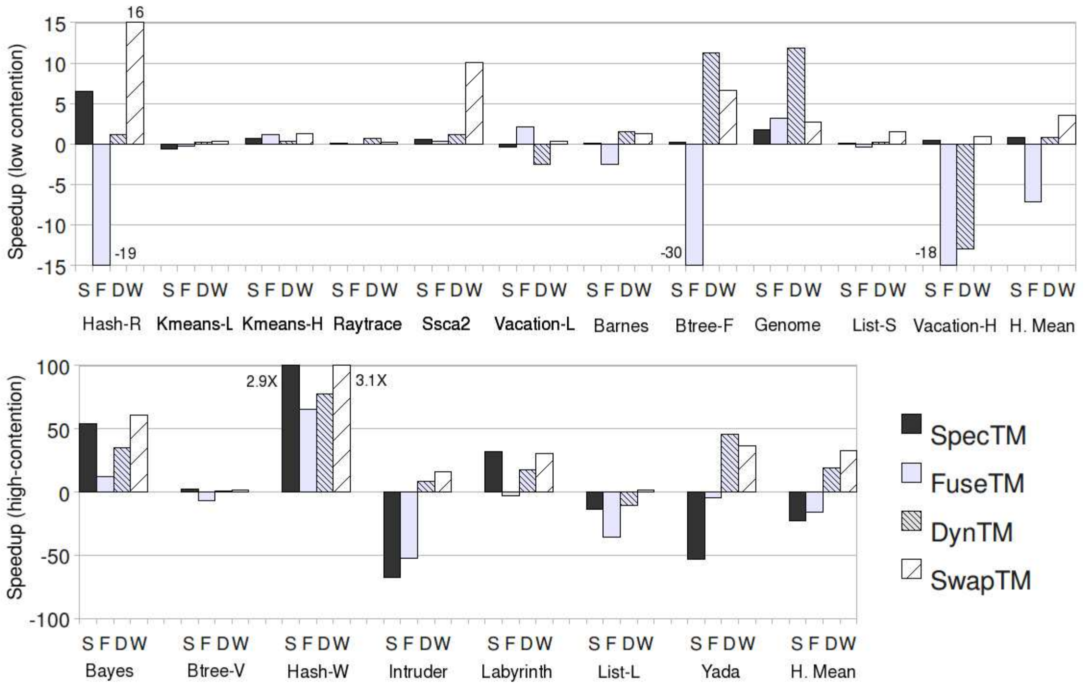


Figure 6.15: Speedup over best-performing fixed-policy HTM of low- and medium-contention (top, 32 threads) and high-contention (bottom, 16 threads) TM applications under SPECTM (S), FUSETM (F), DYNTM (D) and SWAPTM (W)

benchmarks. That is the case of *Hash-write* or *Yada*, where fixed-policy HTM systems suffer notable performance pathologies.

Figure 6.15 shows the speedup obtained on dynamic HTM systems (SPECTM, FUSETM, DYNTM and SWAPTM) over the best-performing fixed-policy HTM system (TCC-Loc for low-contention, FASTM-IVM for high-contention applications). In the low-contention scenario, SWAPTM beats DYNTM and SPECTM by a 3% performance improvement, and FUSETM by around a 5%. This is because SWAPTM can take advantage of selective logging to run ahead speculatively without additional aborts—a condition that FUSETM and DYNTM do not satisfy.

On the other hand, in high-contention applications SWAPTM obtains an average speedup of 13% (DYNTM), 51% (FUSETM) and 72% (SPECTM). This happens because some of the benchmarks execute transactions with variable length and contention—this difference on transactional behavior is noticeable even among instances of the same transaction. That is the case for *Bayes*, *Hash-write* or *List-long*. In those complex situations, SWAPTM picks on the run

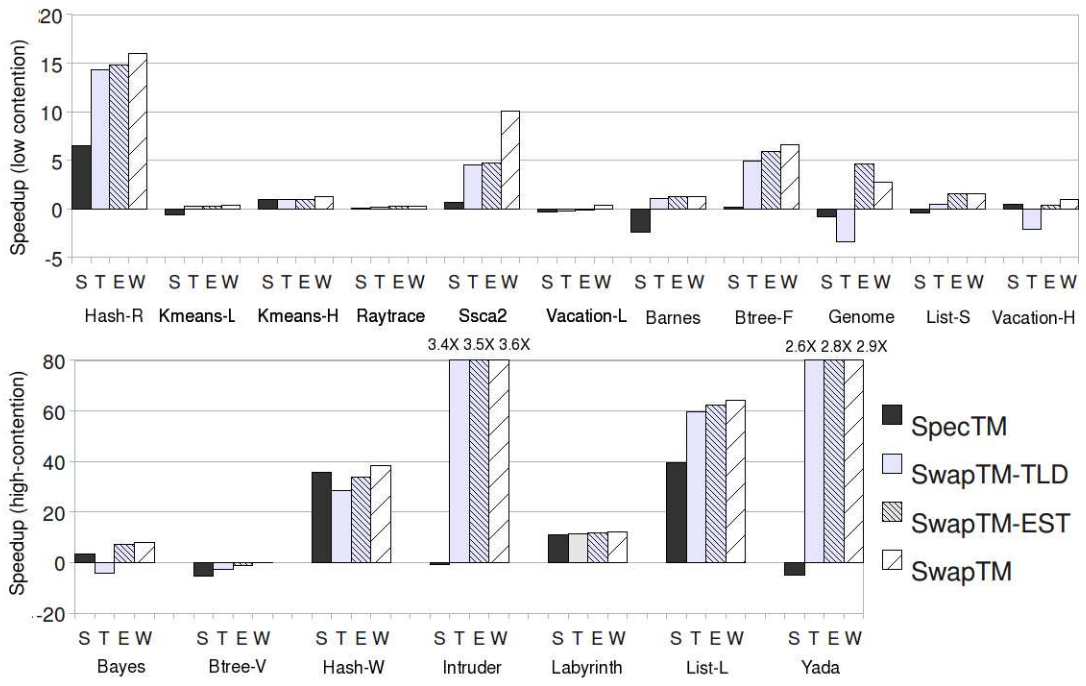


Figure 6.16: Speedup achieved over TCC-Loc in low-contention (top, 32 threads) and high-contention (bottom, 16 threads) applications in SPECTM (S), SWAPTM-TLD (T), SWAPTM-EST (E) and SWAPTM (W)

the best conflict management strategy based on analysis information of each dynamic instance of a transaction. This technique allows SWAPTM to obtain up to 2X speedup over DYNTM in *Hash-write*. Following we present a deeper analysis on SWAPTM strengths.

Fine-grain flexibility. Unlike DYNTM, SWAPTM does not enforce early VM for those transactions that do not fit in the L1 cache. Instead, it relies on SPECTM's logging and engine for keeping safe those speculative data that leave in-core memory space. However, SPECTM lacks flexibility—it executes all the transactions with lazy resolution of conflicts. Figure 6.16 shows the speedup of three different SWAPTM alternatives compared to SPECTM. As it can be seen, having a fixed policy hurts drastically the performance of TM applications that combine different-style transactions (*Bayes*, *List-long* and *Yada*) or manifest dynamic behavior (*e.g.*, *Btree-fix*, *Genome* and *Intruder*). In average, SWAPTM obtains a 72% (3.4%) performance improvement against SPECTM in high-performance (low-performance) benchmarks.

Useful profiling information. Figure 6.16 provides additional information about the adaptive

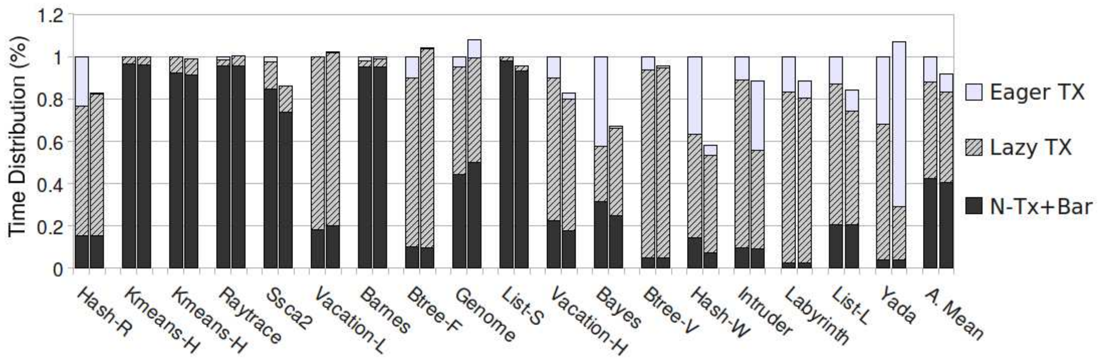


Figure 6.17: DYNTM and SWAPTM execution time of low-contention (left, 32 threads) and high-contention (right, 16 threads) applications distributed by the transactional mode

engine implemented by SWAPTM. The LTD mechanism permits a rapid switch toward eager CM when a transaction is considered large. As a result, SWAPTM-TLD improves on SPECTM in applications with variable-length transactions such as *Hash-read*, *Intruder* or *Yada*. However, moving to eager CM too often or too early may incur a significant overhead in applications with long transactions and read-write conflicts, as it restricts concurrency among threads and imposes backoff. This limitation also affects DYNTM, which must run selected transactions in eager mode from the very beginning. To prevent cascade of stalls we evaluate SWAPTM-EST, an instrument that change to lazy CM when a transaction provokes starvation. On top of that system we build SWAPTM, which moves to lazy CM when a cycle of stalled transactions occurs. As it can be seen, going back to lazy mode has some benefits in applications with read-write conflicts like *Genome* or *List-long*.

Great adaptability. The sum of the previously described mechanisms offers a fast in-time reaction to transactional events. This profit is significant in applications where instances of the same transaction produce an uncertain behavior (*e.g.*, *Bayes* and *Labyrinth*), where SWAPTM obtains better performance than DYNTM at less hardware cost—SWAPTM does not require the TMS predictor.

Effective combination of CM policies. As it is shown in Figure 6.17, SWAPTM distributes wisely eager and lazy execution: eager transactions are not mandatory for large transactions nor require conservative re-execution, while lazy transactions do not abort due to overflows and can prevent read-write conflicts that end up as aborts. In fact, in some benchmarks like *Ssca2* the on-fly swapping mechanism balances the speculative execution flow, which reduces the barrier cycles and overall execution time.

6.5 Results Roadmap: A General View

This section reviews the performance of the HTM systems enclosed on this thesis—*i.e.*, FASTM under the *cycle* policy, FUSETM, SPECTM, DYNTM and SWAPTM—and compares their performance against our reference HTM systems—LogTM-SE and TCC-Dist. The next paragraphs expose how the intrinsic properties of each benchmark affects its performance and scalability. We grouped the benchmarks according to its level of contention (Figure 6.18 for low contention, Figure 6.19 for medium contention, Figure 6.20 for high contention), and we present speedup numbers over sequential execution after running TM applications with up to 32 threads.

6.5.1 Low-contention Applications

Hash-read. In this application, parallel threads execute most of the time searches in a large shared hash table, although they can also insert or remove data randomly. This kind of applications normally perform better in lazy environments, like SPECTM or TCC-Dist. FUSETM cannot take advantage of lazy capabilities because some transactions overflow the L1 cache, and thus they need to be aborted and re-executed in eager mode. Dynamic HTM systems—especially SWAPTM, which lazy mode is not bounded by transactions' size—can rapidly adapt a more conservative conflict policy when two threads update shared data, a strategy that reports better performance. Nonetheless, eager HTM systems keep scaling with many-threads, as *Hash-Read* is a low-contention benchmark.

Kmeans-low. This workload is the only one of eighteen where all HTM systems obtain similar (extremely good) scalability. This is because *Kmeans-low* mostly executes non-transactional embarrassingly parallel code, and thus concurrent threads do not need to synchronize.

Kmeans-high. Like *Kmeans-low*, this STAMP benchmark runs in its majority outside transactional blocks. However, the few (commonly small) transactions that it executes may collide, producing aborts. LogTM-SE recovers the state by software, adding overheads that expose

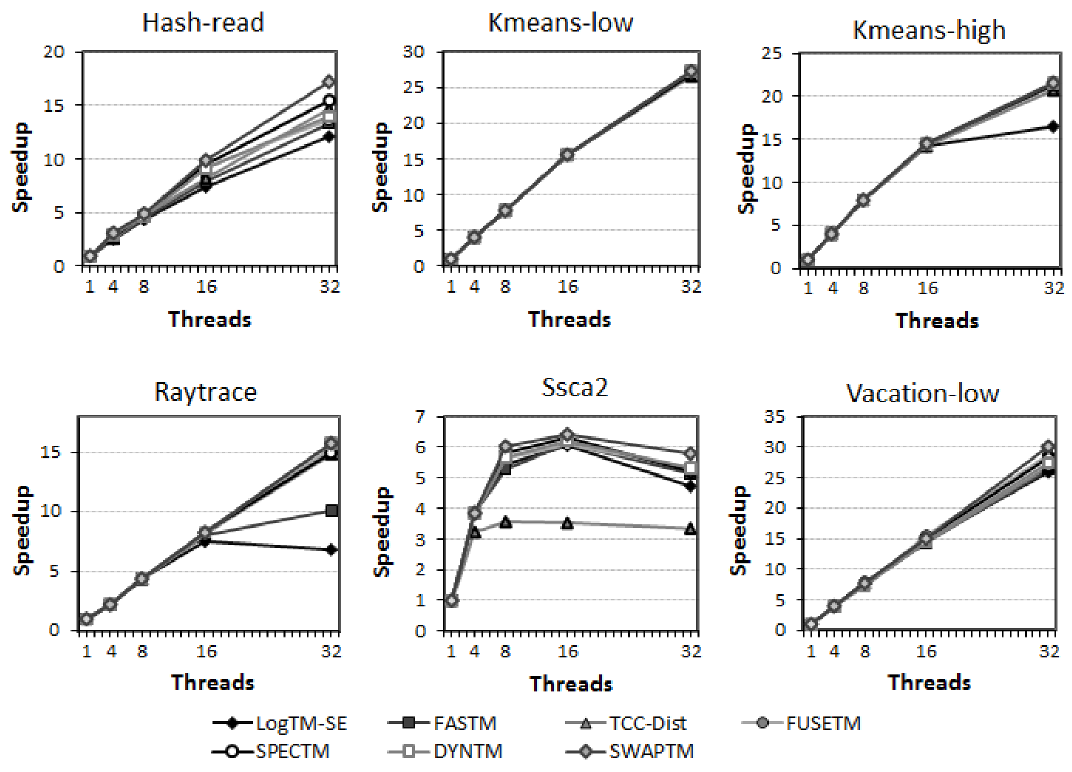


Figure 6.18: Scalability analysis of HTM systems on low-contention applications

more time transactions to conflicts. In contrast, FASTM and the rest of the proposals are not challenged by *Kmeans-high* because they perform an (almost) immediate restoration of the validated state.

Raytrace. This application suffers considerably delays when it is executed under 32 threads with eager conflict management. The reason behind this limitation lies on the high volume of read-write conflicts risen at runtime, and the poor job that eager schemes do when they must spread contention. As most of the transactions are small, FUSETM is able to run them in fast lazy mode, behaving like SPECTM. Dynamic approaches execute almost all transactional instances in lazy mode, here the explanation why they perform like the ideal TCC-Dist in *Raytrace*.

Ssca2. This benchmark consists on multiple kernels accessing a single data structure representing a weighted, directed multigraph. It executes tiny transactions that do not usually conflict. This behavior affects negatively TCC-Dist, which commit protocol collapses the network and the directory. Although this benchmark presents a really low conflict rate, it does not scale with

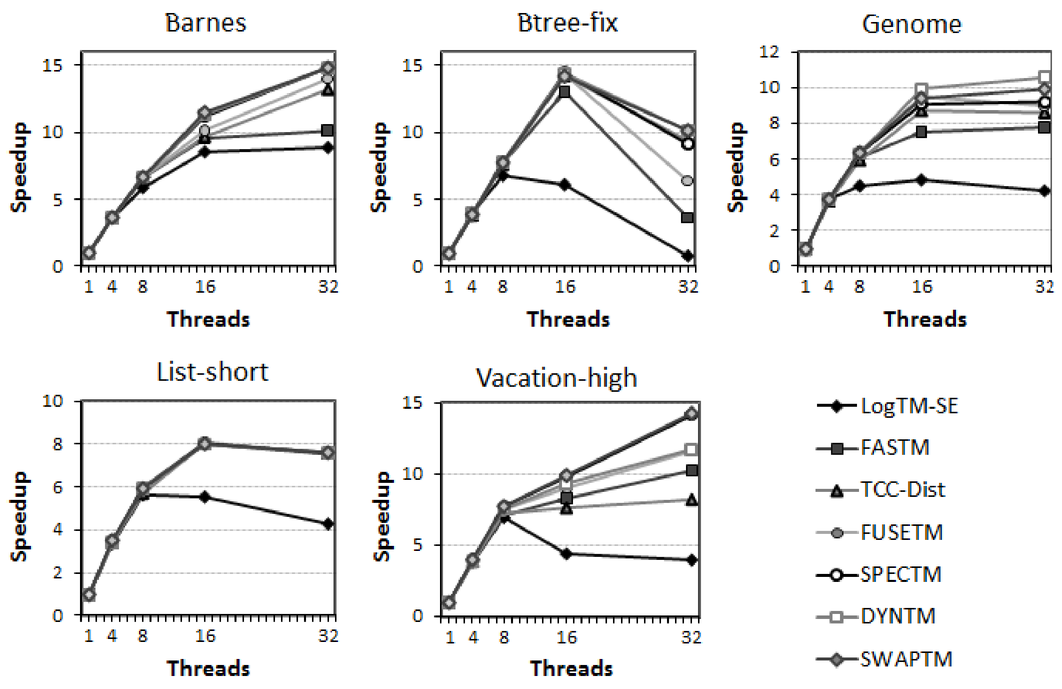


Figure 6.19: Scalability analysis of HTM systems on medium-contention applications

32 threads, as great part of the execution time is sequential—all except one threads wait most of the times in barriers.

Vacation-low. In this low-contention configuration, *Vacation* searches on large regions of shared data structures, and at the end writes few data in another list. Hence, few transactions collide, allowing almost linear scalability. Only with 32 threads the dynamic HTM versions outperform the rest, as the can operate smartly on these rare occasions where performance pathologies appear.

6.5.2 Medium-contention Applications

Barnes. This application is another example of the importance of having fast commits and deferred resolution of conflicts in applications with small transactions. Eager HTM systems (Logtm-SE and FASTM) do not scale when *Barnes* is executed with many threads, whereas lazy HTM systems keep extracting parallelism. Like in *Ssca2*, TCC-Dist suffers delays when it has to commit many short transactions, given that it saturates shared resources.

Btree-fix. In this configuration, the workload computed data in a shared binary tree after doing a *homogeneous* computation—*i.e.*, the time computing the data does not vary. While all but LogTM-SE approaches perform similarly with 16 threads or less, with 32 threads the performance drops down, especially on eager approaches. The reason behind this behavior is that accessing a data structure when it is empty causes lots of conflicts and data transfers in the CMP, lowering the efficiency of the application. An execution of the benchmark with more transactions (and thus populating the tree faster) should increase the scalability of the workload for all HTM systems.

Genome. This workload is a good reference point to study the performance of HTM systems in applications with phase-changes. As it can be seen, both DynTM and SwapTM do a good pursuit dealing with conflicts when the contention level varies through time. Although both approaches eliminate the impact of conflicts at runtime, the sequential parts of the application bound the scalability of the program in many-threaded executions.

List-short. This benchmark is similar to *Kmeans*: large parallel (non-transactional) sections of code combined with short transactions, with two key differences. First, simultaneous threads share most of the data, making the CMP configuration (memory hierarchy, distributed directory, network topology, *etc.*) the main bottleneck to achieve good scalability. Second, abort recovery is critical, here the reason for the inefficiency of LogTM-SE. Note that, in contrast to *Ssca2*, small transactions are executed in the middle of parallel phases. This fact allows TCC-Dist to hide its slow commits, as it is rare that two transactions commit at the same time.

Vacation-high. This configuration of *Vacation* is a clear example of the importance of enabling speculation even in the case of overflow. On cache evictions, SpecTM and SwapTM are able to keep deferring the resolution of conflicts until commit time (or until the system suggests a wiser conflict management decision), avoiding some read-write conflicts that arise on many-threaded executions. TCC-Dist must execute those transactions lazily until the end, which sometimes provokes the re-execution of the older (and thus critical) transaction. This may cause thread unbalance in the application.

6.5.3 High-contention Applications

Bayes. This workload, which consists on building a belief network, is very challenging for modern HTM systems. It executes different-type transactions, some of them huge. Moreover,

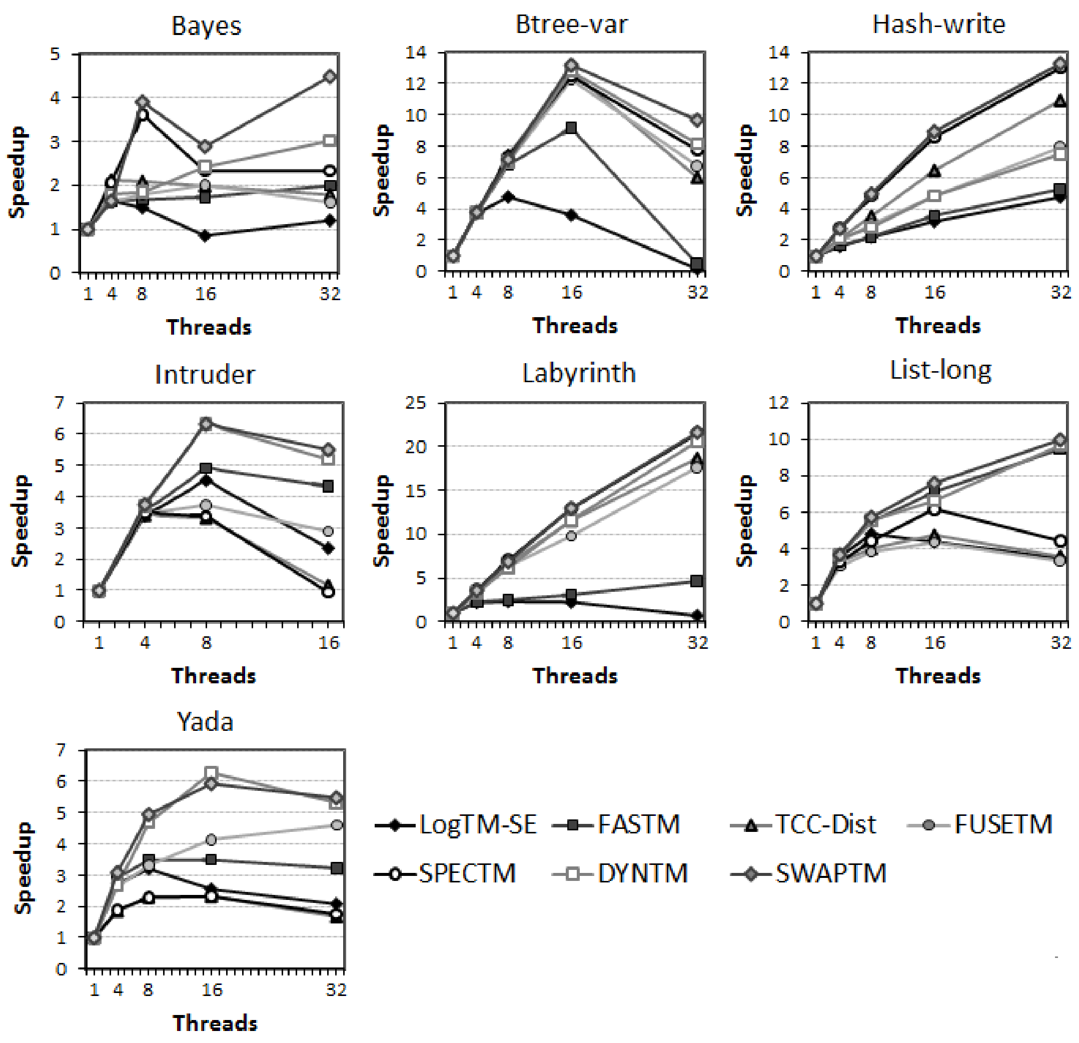


Figure 6.20: Scalability analysis of HTM systems on high-contention applications

as it can be seen from the data in Figure 6.20, *Bayes* suffers performance degradation at 16 threads, but it recovers most of it at 32 threads. SWAPTM is the best of all HTM systems, and it outperforms DYNTM because it does not rely on an erratic predictor to select the transactional execution mode. As FASTM or SPECTM do not have to abort in case of transactional eviction, they provide better conduct than FUSETM.

Btree-var. Like the fixed version of *Btree*, this microbenchmark do not scale beyond 16 threads. Eager HTM systems like FASTM or LogTM-SE must be inflexible in the way of resolving conflicts, which may prevent the forward progress of critical transactions when they collide with ones with lower priority. Speculative systems, instead, are able to run-ahead in case of conflict. Such freedom helps lazy HTMs like TCC-Dist to be more effective with respect to eager-based HTMs. In contrast to *Btree-fix*, here most transactions fit on the L1 cache, therefore FUSETM is not affected by aborts caused by overflows. The main limitation of dynamic approaches is that *Btree-var* executes transactions with different characteristics with a random pattern, which makes difficult for DYNTM history-based predictor to identify the most profitable execution mode.

Hash-write. This microbenchmark executes large transactions with opposed contention characteristics: some transactions are parallel, while the others are mostly serial. The main limitation of fixed policy systems (TCC-Dist or FASTM) is their lack of flexibility, while some dynamic HTM systems (FUSETM and DYNTM) bound their performance because their execution modes are tied to L1 buffering space. As a result, partly (SPECTM) or fully (SWAPTM) adaptive and non-restrictive HTM systems improve on previous approaches, obtaining more than 10X speedup over FASTM with 32 threads.

Intruder. This is possible the poorest scaling workload from the STAMP benchmark suite, especially when it is executed with *simulation* small-size inputs—other inputs suggested for STM systems achieve more scalability [16]. The maximum performance is reached with 8 threads (around 6X speedup over sequential execution when run in dynamic approaches). Note that *Intruder* is strongly affected by the *starvation of the older* pathology [14] in lazy environments like TCC-Dist or SPECTM, working worse than FASTM or even LogTM-SE.

Labyrinth. This workload exemplifies the importance of having fast abort recovery and flexible conflict management. Apparently, *Labyrinth* is a high-contention workload, especially at the beginning of its execution. Inefficient eager policies may delay that phase during too much time, putting obstacles on the path of pressing transactions. Software abort increases the conflict window, adding more risk of serialization. As a matter of fact, SWAPTM obtains 20X over sequential execution when it is run with 32 threads, while LogTM-SE barely improves single-threaded execution.

List-long. Updating shared lists with data computed during several cycles can be an important issue for transactions that resolve conflicts at commit time—TCC-Dist performs worse than low-cost LogTM-SE. Dynamic (DYNTM and SWAPTM) and eager high-performing (FASTM) systems can preserve useful work of *List-long* by stopping conflicting transactions when they are close to their end, minimizing the impact of re-executing after introducing inconsistencies.

Yada. This is another high-contention workload that do not scale beyond 16 threads. It combines small and large transactions with variable contention: such properties impose hard constraints on fixed-policy HTM systems. In this application, FUSETM takes advantage of the eager mode of execution for long transactions, and thus it outperforms unique-mode HTMs like SPECTM. Dynamic HTM systems are better-suited than non-adaptive systems (either lazy-based TCC-Dist or eager-based FASTM), as they can adopt the most beneficial execution mode according the characteristics of *Yada*'s transactions.

6.6 Related Work on Contention-Aware HTM Systems

Most HTM proposals fit in the eager/lazy HTM categories—the last two chapters review in detail both HTM designs. Nonetheless, there are some hardware-assisted TM designs that establish novel conflict management techniques to favor concurrency and attest high performance. In this section, we revise those designs that reduce contention by applying software-managed conflict management policies, tracking dependences between transactions, predicting speculative values or performing partial re-executions.

Conventional HTM systems implement a *two-phase-locking* (2PL [32]) algorithm that serializes the execution of conflicting transactions. Thus, in case of collisions, one of the conflicting transactions has to abort or delay its execution until the conflict disappears. This may become a bottleneck in applications with a high volume of sharing data but a small conflicting set. *Conflict-serializability* (CS [6]) is a more relaxed algorithm that allows concurrency between conflicting threads by tracking (and ordering) data dependences.

Ramadan *et al.* implemented such algorithm in hardware through a Dependence-Aware HTM (DATM [99]) system. DATM accepts more interleaving than 2PL by forwarding non-committed values using a non-standard coherence protocol. This approach permits overcoming *direct* (non-cyclical) WAW (write-after-write) and RAW (read-after-write) conflicts, but cores must keep a single, updated order between transactions during the whole execution. If a conflicting cycle appears, at least one transaction is aborted and the global order is re-calculated. Moreover, short transactions may have to wait for longer transactions to commit. To reduce the

HTM System	VM Strategy	Hardware Support	CM Strategy	Fixed Policy?	Versatility Granularity
DATM [99]	Late	Forwarding L1 protocol	Speculate with WAW and RAW	Yes	Application
WarTx [122]	Early	Tx Store Buffer, SW logging	Speculate with WAR conflicts	Yes	Application
FlexTM [112]	Late	Tx L1 cache, SW support	Lazy or Eager	No[2]	Application
FASTM	Early	TMESI L1 protocol	Eager	Yes	Application
FUSETM	Late Early (OV Tx)	UTCP L1 protocol	Lazy or Eager	Yes[3]	Overflowing Transactions
SPECTM	Late or Early (OV data)	UTCP L1 protocol, Selective log	Lazy	Yes	Application
DYNTM	Late or Early	UTCP L1 protocol, TMS predictor	Lazy or Eager	No	Individual Tx Instances
SWAPTM	Late or Early (OV data)	SPECTM-like & profiling HW	Lazy or Eager	No	Individual Tx Instances

Table 6.2: Data VM and CM characteristics of high-performance HTM systems

hardware complexity of DATM, Utke *et al.* assigned a *serializability ordering number* (SON) to each committing transaction [7]. Their system implements late version management and tracks the read history and conflicts in hardware tables. At commit time, negative acknowledgements are broadcast to enforce global seralizable order between conflicting transactions. Pant *et al.* also showed how to reduce the hardware overheads by using advanced concurrency monitoring techniques [90].

Titos *et al.* followed a distinct approach to survive WAR (write-after-read) conflicts in a log-based HTM system [122]. While non-conflicting writes store transactional modification in-place, conflicting writers maintain the speculative state in a specific gated store buffer until *older* readers commit or abort. Note that this mechanism imposes a global order between in-

[2] in FlexTM, the programmer decides in which mode of execution a transaction is going to run
[3] although it offers two modes of execution, the eager-mode only is operative for overflowing transactions

flight transactions. Again, a cycle between conflicting transactions requires, at least, an abort to break the cyclic dependence. Unfortunately, all proposals impose a strict order between conflicting transactions and can only eliminate *acyclic* dependences. Thus, these HTM systems experiment the same issues as conventional HTM systems when they execute transactions with crossed conflicts, which are common in typical transactional workloads [16].

In order to introduce some flexibility to TM systems, Shriraman *et al.* proposed FlexTM [112], a hybrid implementation [31, 63] that decouples conflict detection from conflict resolution by tracking transactional violations eagerly and delegating their resolution to the software. In such way, the system can operate either eagerly (resolving the conflict at the moment that it is produced) or lazily (resolving the conflict at commit time). Nonetheless, the choice has to be applied for the entire application. For data versioning, it readjusts the underlying *late* VM support from RTM [113] by adding two states to a typical MESI protocol, which hold transactionally written and read lines. This buffering capability is complemented with signatures to summarize accesses within transactions and a hash structure, called Overflow Table, which must be accessed by software to perform lookups on cache misses and to ensure permanent commits.

FlexTM's dual-mode system permits the programmer to decide the conflict management scheme for the entire application (eager or lazy), but it requires (i) software decisions to resolve conflicts, (ii) complex hardware to buffer transactional overflowed data (because it uses late VM) and (iii) software commit arbitration for lazy transactions. What is more, FlexTM applies the same conflict management policy for the whole execution, being more restrictive than an HTM system that dynamically adopts the policy at the granularity of a transaction—something that DYNTM and SWAPTM can do.

Two other HTM systems appeared recently that also mix concepts from eager and lazy approaches. LagerTM [132] retains few lines privately in a gated store buffer to emulate lazy execution for conflicting lines, while other memory accesses are executed under eager semantics. A local-accessed structure informs each core which lines must be kept hidden in the store buffer (in the common case, those that conflict frequent). The store buffer must be drained at commit time. Similarly, ZEBRA [123] modifies the coherence protocol engine to redirect those speculative updates that are marked as contended in the L1 cache to a special buffer (like late

VM systems). Non-contending data can be safely moved towards the memory hierarchy because old values are maintained on the side (like early VM systems). Note that both proposals introduce small data structures to keep the pre-transactional and/or the speculative state. Those private buffers can easily be overflowed, reducing the potential performance gain in applications with large memory footprints. Instead, DYNTM and SWAPTM modify bigger, already existing buffers to permit flexible policies for any kind of transaction.

Table 6.2 shows the differences between enhanced HTM systems and our proposals. As we can see, our main contribution lays on the possibility of adapting the system at runtime at the granularity of a transaction, something that other HTM systems cannot do. The non-dependency on software, the version management flexibility or the feasibility of the design are also a plus.

Another way to deal with high-contention situations is to guess the value generated by transactions that conflict. Tabba *et al.* speculated in their Transactional Value Prediction (TVP [120]) implementation with false sharing conflicts—those caused by cache-line granularity—by using stale data cache lines on transactional loads and validating those loads (and also those stores from whom the processor does not have exclusive permissions) at commit time.

Value Prediction Transactional Memory (VP-TM [89]) implements a memory-level predictor over a log-based HTM system that attempts to anticipate the future value of a conflicting line. The predictor assumes a well-known pattern in commonly updated shared variables—*e.g.*, a counter that is always incremented—and supplies the generated value of the line to the processor, which has to validate the correctness of the line when the conflict disappears—this happens when the clashing transaction commits or aborts. Notice that value prediction does not impose an order between conflicting transactions, being more flexible than CS-based HTM systems. Of course predicting the correct value for a line is not straightforward.

Nesting transactions (either close or open) can be used to reduce the wasted work on aborts. Many HTM proposals can be extended for supporting different forms of nesting [43]. An alternative way to minimize discarded work consists on using automatic *intermediate* checkpoints [126] to restart transactions from a convenient snapshot rather than from the very beginning of the transaction. These techniques are orthogonal to dynamic conflict management, and we do not see any restriction for using them to improve further the performance of DYNTM or SWAPTM.

6.7 Conclusions

In this chapter we have presented DYNTM and SWAPTM, two fully flexible HTM systems that adapt their version and conflict management strategies according to the characteristics of each individual instance of a transaction executed in the system. This versatility allows DYNTM and SWAPTM to take smart decisions to resolve complex conflicts, something that modern HTM systems that fix the conflict management strategy at design time cannot do.

DYNTM extends the FUSETM system with the Transactional Mode Selector (TMS), a history-based predictor that takes advantage of the flexibility offered by the underlying hardware to decide the best execution mode for each transaction at runtime. The predictor executes those transactions that tend to overflow the L1 cache or those transactions that are prone to abort multiple times in eager mode, which saves computational work and minimizes the abort rate. The rest of transactions are executed lazily to favor concurrency in the system. DYNTM also gets benefit from a novel, high-performance policy to efficiently resolve eager-lazy conflicts. In high-contention applications, DYNTM obtains an average speedup of 19% over the best (idealized) HTM system that employ fixed version and conflict management mechanisms, a 24% speedup over FUSETM and a 12% speedup over an HTM system that applies static conflict management policies.

While DYNTM re-executes in eager mode those transactions that exceed the L1 cache, SWAPTM handles on the fly the eviction, removing inconsistencies if necessary. This mechanism hybrid data versioning technique offers more flexibility than DYNTM conflict management approach. For instance, SWAPTM takes out the predictor from the system and uses instead profiling techniques to modify at any point in time the behavior of each individual instance of a transaction. As a result, SWAPTM obtains an average speedup of 11% over DYNTM.

Chapter 7

Conclusions

The consolidation in the multicore era offers numerous opportunities to develop powerful applications in the near future. Transactional Memory proposes a simple and efficient programming model to enhance the performance of parallel software without sacrifying ease of use. This thesis examines different alternatives to implement a high-performance hardware-assisted TM system using architectural on-chip support. In this chapter, we first expose a summary of the contributions enclosed in this dissertation and then we follow discussing future work.

7.1 Summary

In this thesis, we address the latent problems associated with modern HTM systems by presenting five advanced HTM designs that require distinct transactional mechanisms.

FASTM. We propose a low-cost *eager* HTM system that modifies the L1 cache controller and the coherence protocol to eliminate most of the software aborts, which in turn minimizes the number of conflicts and favors overall concurrency. On the one hand, FASTM forces write-backs to the L2 cache on coherence transitions, which transparently guarantees that the non-speculative state is pinned down in the upper levels of the memory hierarchy. On the other hand, a software log is kept on the side with a copy of these values, which permits in-place transactional replacements without additional actions. Hence, FASTM takes advantage of a *hybrid* version management mechanism that collects the best of previous (early or late) data versioning proposals. This HTM system can be further improved by coupling the original proposal with selective logging or wake-up notification.

FuseTM and SpecTM. We rethink the concept of speculative transactions by presenting FuseTM, a non-invasive *lazy* HTM system that offers a unified framework where different-mode transactions can be simultaneously executed. In FuseTM, transactions that fit in the L1 cache defer the resolution of conflicts at commit time, although it detects (and tracks) collisions as soon as they are produced. This infrastructure permits the implementation of *local* commits that avoid long delays when the system executes short transactions. What is more, by restarting in eager mode those transaction that overflow the L1 cache, the FuseTM system saves a considerable amount of on-chip area—previous lazy HTM systems require additional hardware to maintain the overflowing state. SpecTM extends FuseTM to support deferred resolution of conflicts for most of the lines of a transaction, independently of its size.

DynTM and SwapTM. We break with the assumption that HTM system must fix transactional mechanisms at design time by introducing two truly flexible HTM systems: DynTM and SwapTM. The former determines the best-suited execution mode for each individual instance of a transaction by recording past information of previously committed or aborted instances. The latter decouples conflict management from version management in order to switch the execution mode on the fly. Cutting off the dependency on the conflict and version management strategy during the whole execution enables high concurrency on applications with heterogeneous transactions—those that present variable sizes and different levels of contention—and good performance on applications that carry a dynamic behavior.

7.2 Future Work

The wide versatility of the HTM systems presented in this dissertation opens new venues for future TM research.

7.2.1 Eager HTM Systems

FasTM is only the first step to bridge the gap between early and late VM implementations. Nonetheless, we believe that there is plenty of work that can help to improve further eager HTM systems. In this thesis, we have seen that FasTM reduces the pressure on signatures, increasing their fidelity and reducing the overhead of false positives. However, the implementation of this mechanism is still critical when large transactions are executed. Thus, it would be interesting to study new methods to track big read and write sets, such as hierarchical or asymmetric configurations.

This thesis also demonstrates that many-threaded executions suffer considerable delays when running high-contention applications, even after applying advanced conflict resolution policies. Wake-up notification is a good strategy to save global power on those situations. This power budget could be used to accelerate critical transactions (*e.g.*, by applying DVFS techniques [61]) and reduce the conflict window. Similarly, it would be attractive to study the impact of coupling an Asymmetric CMP [117] with TM support, which would permit critical transactions to speed up their execution as soon as they are moved to the faster core.

7.2.2 Lazy HTM Systems

This thesis shows how a lazy HTM system can be built using eager-like hardware. To achieve our goal, we have assumed simple, in-order cores to facilitate the comprehension of the design. Nonetheless, the industry is manufacturing out-of-order (OoO) processors to increase ILP. This core configuration introduces several microarchitectural structures (*e.g.*, store buffers) or events (*e.g.*, branch misspeculation) that the transactional mechanisms must be aware of. Although few OoO architectures with TM support have been explored (see the Rock prototype [23] for more info), all of them assumed an eager HTM scheme. Thus, it would be nice to see how a lazy HTM system can be integrated in a CMP formed by OoO processors.

One of the advantages of using store buffers is that the system can implement relaxed consistency models. Inside transactions, it is enough to satisfy transactional consistency (of course, register dependences must be preserved). In such weak models, memory accesses can be reordered at free will. It would be interesting to research how HTM systems can take advantage of this flexibility—delaying offending memory accesses at the end of a transaction may improve the overlap of conflicting atomic blocks.

7.2.3 Dynamic HTM Systems

The dynamically adaptive HTM systems presented through this thesis provide an important framework for future HTM studies. A lot of interesting work on conflict management on the face of adaptability is possible. A reasonable starting point could be extending the two available execution modes with others that employ more aggressive or conservative features. The conflict resolution policy between different-mode transactions can also be revised to establish distinct priority levels to promote critical transactions.

More sophisticated mechanisms for deciding the right execution mode could be devised as well. For instance, the DYNTM predictor can be expanded with additional parameters in order to figure out with high accuracy how future instances of a transaction will behave. Improvements on the coherence protocol, reductions on complexity, implications to the TM runtime, and others are also topics that could be revisited on the type of systems enabled by DYNTM or SWAPTM.

Bibliography

[1] http://www.cs.wisc.edu/trans-memory/biblio/.

[2] *TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors*, Aug 2002.

[3] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *Procs of the 14th Symp on Principles and Practice of Parallel Programming*, Feb 2009.

[4] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proc of the Intl Conf on Programming Language Design and Implementation*, Jun 2006.

[5] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Procs of the 11th Intl Symp on High-Performance Computer Architecture*, Feb 2005.

[6] Utku Aydonat and Tarek Abdelrahman. Serializability of Transactions in Software Transactional Memory. In *Procs of the 3rd Workshop on Transactional Computing*, Feb 2008.

[7] Utku Aydonat and Tarek Abdelrahman. Hardware Support for Relaxed Concurrency Control in Transactional Memory Systems. In *Procs of the 43rd Intl Symp on Microarchitecture*, Dec 2010.

[8] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly Atomic Hybrid Transactional Memory . In *Procs of the 35th Intl Symp on Computer Architecture*, Jun 2008.

[9] Burton H. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13:7, 1970.

[10] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making The Fast Case Common And The Uncommon Case Simple In Unbounded Transactional Memory. In *Procs of the 34th Intl Symp on Computer Architecture*, Jun 2007.

[11] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *Procs of the 36th Intl Symp on Computer Architecture*, 2009.

[12] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *Procs of the 35th Intl Symp on Computer Architecture*, Jun 2008.

[13] Jayaram Bobba, Marc Lupon, Mark D. Hill, and David A. Wood. Safe and Efficient Supervised Memory Systems. In *Procs of the 17th Intl Symp on High-Performance Computer Architecture*, Feb 2011.

[14] Jayaram Bobba, Kevin E. Moore, Luke Yen, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Procs of the 34th Intl Symp on Computer Architecture*, Jun 2007.

[15] Jayaram Bobba, Weiwei Xiong, Luke Yen, Mark D. Hill, and David A. Wood. StealthTest: Low Overhead Online Software Testing using Transactional Memory. In *Procs of the 18th Intl Conf on Parallel Architectures and Compilation Techniques*, Sep 2009.

[16] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Procs of The IEEE Intl Symp on Workload Characterization*, Sep 2008.

[17] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Procs of the 34th Intl Symp on Computer Architecture*, Jun 2007.

[18] Calin Cascaval, Colin Blundell, Maged Micheal, Harold Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why is it only a research toy? *Communications of the ACM*, 51(11):40–46, Nov 2008.

[19] Michel Cekleov and Michel Dubois. Virtual-address caches, part 2: Multiprocessor issues. *IEEE Micro*, 17, Nov 1997.

[20] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Procs of the 33th Intl Symp on Computer Architecture*, Jun 2006.

[21] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. In *Procs of the 34th Intl Symp on Computer Architecture*, June 2007.

[22] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *Procs of the 13th Intl Symp on High-Performance Computer Architecture*, Feb 2007.

[23] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 29(2), Apr 2009.

[24] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded Page-Based Transactional Memory. In *Procs of the 12th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, Mar 2006.

[25] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in Transactional Memory Virtualization. In *Procs of the 12th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, Oct 2006.

[26] JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *Procs of the 12th Intl Symp on High-Performance Computer Architecture*, Feb 2006.

[27] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, Dan Grossman, and David Christie. ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory. In *Procs of the 43rd Intl Symp on Microarchitecture*, Dec 2010.

[28] Pat Conway and Bill Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27:10–21, Mar 2007.

[29] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[30] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Procs of the 15th Symp on Principles and Practice of Parallel Programming*, Jan 2010.

[31] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid Transactional Memory. In *Procs of the 12th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, Oct 2006.

[32] Dave Dice, Yossi Lev, Mark Moir, and Dan Nussbaum. Transactional Locking II. In *Procs of the 14th Intl Conf on Distributed Computing*, Sept 2006.

[33] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Procs of the 14th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, Mar 2009.

[34] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In Shlomi Dolev, editor, *Distributed Computing*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006.

[35] Aleksandar Dragojević, Rachid Guerraoui, and Michael Kapalka. Stretching Transactional Memory. In *Procs of the 2009 Intl Conf on Programming Language Design and Implementation*, Jun 2009.

[36] Stijn Eyerman and Lieven Eeckhout. Modeling Critical Sections in Amdahl's Law and its Implications for Multicore Design. In *Procs of the 37th Intl Symp on Computer Architecture*, 2010.

[37] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Procs of the 13th Intl Symp on Principles and Practice of Parallel Programming*, Feb 2008.

[38] Tim Harris Osman Unsal Adrián Cristal Ibrahim Hur Mateo Valero Ferad Zyulkyarov, Srdjan Stipic. Discovering and understanding performance bottlenecks in transactional applications. In *Procs of the 19th Intl Conf on Parallel Architectures and Compilation Techniques*, Sep 2010.

[39] James R. Goodman. Coherency for multiprocessor virtual address caches. In *Procs of the Intl Conf on Architectual Support for Programming Languages and Operating Systems*, 1987.

[40] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An Efficient Software Transactional Memory Using Commit-time Invalidation. In *Procs of the Intl Symp on Code Generation and Optimization*, Apr 2010.

[41] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[42] Shantanu Gupta, Florin Sultan, Srihari Cadambi, Franjo Ivancic, and Martin Rotteler. Using Hardware Transactional Memory for Data Race Detection. In *Procs of the 23rd Intl Symp on Parallel and Distributed Processing Symposium*, May 2009.

[43] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Composing First-Class Transactions. *ACM Transactions on Programming Languages and Systems*, 16, 1994.

[44] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with Transactional Coherence and Consistency (TCC). In *Procs of the 11th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, Oct 2004.

[45] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *Procs of the 31st Intl Symp on Computer Architecture*, Jun 2004.

[46] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, Jun 2010.

[47] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing Memory Transactions. In *Procs of the Intl Conf on Programming Language Design and Implementation*, Jun 2006.

[48] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Procs of the 22nd Symp on Principles of Distributed Computing*, Jul 2003.

[49] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Procs of the 20th Intl Symp on Computer Architecture*, May 1993.

[50] Enric Herrero, José González, and Ramon Canal. Elastic Cooperative Caching: an Autonomous Dynamically Adaptive Memory Hierarchy for Chip Multiprocessors. In *Procs of the 37th Intl Symp on Computer Architecture*, Jun 2010.

[51] Mark D. Hill, Derek Hower, Kevin E. Moore, Michael M. Swift, Haris Volos, and David A. Wood. A Case for Deconstructing Hardware Transactional Memory Systems. In *Programming Models for Ubiquitous Parallelism*, 2007.

[52] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. volume 41, Jul 2008.

[53] Lorin Hochstein, Victor R. Basili, Uzi Vishkin, and John Gilbert. A Pilot Study to Compare Programming Effort for Two Parallel Programming Models. *Journal of System Software*, 81, Nov 2008.

[54] Owen S. Hofmann, Christopher J. Rossbach, and Emmett Witchel. Maximum Benefit from a Minimal HTM. In *Procs. of the 14th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, Mar 2009.

[55] T. Horel and G. Lauterbach. UltraSPARC-III: Designing Third-Generation 64-bit Performance. volume 19, May/Jun 1999.

[56] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, Jan 2008.

[57] Intel. First the Tick, Now the Tock: Next Generation Intel's Microarchitecture (Nehalem). In *http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf*.

[58] S.A.R. Jafri, M. Thottethodi, and T.N. Vijaykumar. LiteTM: Reducing Transactional State Overhead. In *Procs of the 16th International Symposium on High Performance Computer Architecture*, Jan 2010.

[59] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report Technical Report UCRL-97663, Nov 1987.

[60] Satish Narayanasamy Jie Yu. Tolerating Concurrency Bugs Using Transactions as Lifeguards. In *Procs of the 43rd Intl Symp on Microarchitecture*, Dec 2010.

[61] Wonyoung Kim, Meeta Sharma Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *Procs. of the 14th Intl Symp on High-Performance Computer Architecture*, Feb 2008.

[62] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[63] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid Transactional Memory. In *Procs of the 11th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming*, Mar 2006.

[64] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

[65] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased Transactional Memory. In *Procs of the 2nd Workshop on Transactional Computing*, Aug 2007.

[66] Sean Lie. Hardware Support for Unbounded Transactional Memory. Master's thesis, May 2004. Massachusetts Institute of Technology.

[67] Javier Lira, Carlos Molina, and Antonio González. The Auction: Optimizing Banks Usage in Non-Uniform Cache Architectures. In *Procs of the 24th Intl Conf on Supercomputing*, Jun 2010.

[68] David B. Lomet. Process Structuing, Synchronization and Recovery Using Atomic Actions. In *Procs of the Intl Conf on Language Design for Reliable Software*, Mar 1977.

[69] Marc Lupon. Hardware Approaches for Transactional Memory. Master's thesis, Universitat Politècnica de Catalunya, 2008.

[70] Marc Lupon, Grigorios Magklis, and Antonio Gonzalez. Version Management Alternatives for Hardware Transactional Memory. In *Procs. of the 9th MEDEA Workshop on MEmory performance: DEaling with Applications, systems and architecture*, Oct 2008.

[71] Marc Lupon, Grigorios Magklis, and Antonio González. FASTM: A Log-based Hardware Transactional Memory with Fast Abort Recovery. In *Procs of the 18th Intl Conf on Parallel Architectures and Compilation Techniques*, Sep 2009.

[72] Marc Lupon, Grigorios Magklis, and Antonio Gonzalez. A Dynamically Adaptable Hardware Transactional Memory. In *Procs of the 43rd Intl Symp on Microarchitecture*, Dec 2010.

[73] Marc Lupon, Grigorios Magklis, and Antonio González. A High-performing Hardware Transactional Memory with Swapping Execution Modes. Technical Report UPC-DAC-RR-ARCO-2011-6, Universitat Politècnica de Catalunya, 2011.

[74] Marc Lupon, Grigorios Magklis, and Antonio González. A Selective Logging Mechanism for Hardware Transactional Memory. Technical Report UPC-DAC-RR-ARCO-2011-7, Universitat Politècnica de Catalunya, 2011.

[75] Marc Lupon, Grigorios Magklis, and Antonio González. Lightweight Optimizations for Eager Hardware Transactional Memory Systems. Technical Report UPC-DAC-RR-ARCO-2011-5, Universitat Politècnica de Catalunya, 2011.

[76] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35, 2002.

[77] Virendra Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the Overhead of Software Transactional Memory. In *Procs of the 1st Intl Workshop on Transactional Computing*, Mar 2006.

[78] Virendra J. Marathe and Michael L. Scott. Using LL/SC to Simplify Word-based Software Transactional Memory (poster). In *Procs of the 24th Intl Symp on Principles of Distributed Computing*, Jul 2005.

[79] Kevin Moore Mark Moir and Dan Nussbaum. The Adaptive Transactional Memory Test Platform: A Tool for Experimenting with Transactional Code for Rock. In *Procs of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, Feb 2008.

[80] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *ACM SIGARCH Computer Architecture News*, 33, 2005.

[81] Austen McDonald, JaeWoong Chung, D. Carlstrom Brian, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural Semantics for Practical Transactional Memory. In *Procs of the 33th Intl Symp on Computer Architecture*, Jun 2006.

[82] Austen McDonald, JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Brian D. Carlstrom, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Characterization of TCC on Chip-Multiprocessors. In *Procs of the 14th Intl Conf on Parallel Architectures and Compilation Techniques*, Sep 2005.

[83] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahkle. Parallelizing Sequential Applications on Commodity Hardware Using a Low-Cost Software Transactional Memory. In *Procs of the 2009 Intl Conf on Programming Language Design and Implementation*, Jun 2009.

[84] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *Procs of the 12th Intl Symp on High-Performance Computer Architecture*, Feb 2006.

[85] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting Nested Transactional Memory in LogTM. In *Procs of the 12th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, Oct 2006.

[86] Negi, A. and Waliullah, M.M. and Stenstrom, P. LV*: A Low Complexity Lazy Versioning HTM Infrastructure. In *Procs of the Intl Conf on Embedded Computer Systems*, Jul 2010.

[87] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for a Single-Chip Multiprocessor. In *Procs of the 7th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, Oct 1996.

[88] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An Optimal Bloom Filter Replacement. In *Procs of the 16th Intl Symp on Discrete Algorithms*, 2005.

[89] Salil Pant and Greg Byrd. A Case for using Value Prediction to Improve Performance of Transactional Memory. In *Procs of the 4th Workshop on Transactional Computing*, Feb 2009.

[90] Salil Pant and Gregory Byrd. Limited Early Value Communication to Improve Performance of Transactional Memory. In *Procs of the 23rd Intl Conf on Supercomputing*, Jun 2009.

[91] Leo Porter, Bumyong Choi, and Dean Tullsen. Mapping Out a Path from Hardware Transactional Memory to Speculative Multithreading. In *Procs 18th Intl Conf on Parallel Architectures and Compilation Techniques*, Sep 2009.

[92] Seth H. Pugsley, Manu Awasthi, Niti Madan, Naveen Muralimanohar, and Rajeev Balasubramonian. Scalable and Reliable Communication for Hardware Transactional Memory. In *Procs of the 17th Intl Conf on Parallel Architectures and Compilation Techniques*, Oct 2008.

[93] Xuehai Qian, Wonsun Ahn, and Josep Torrellas. ScalableBulk: Scalable Cache Coherence for Atomic Blocks in a Lazy Environment. In *Procs of the 43rd Intl Symp on Microarchitecture*, Dec 2010.

[94] Ricardo Quislant, Eladio Gutierrez, and Oscar. Plata. Improving Signatures by Locality Exploitation for Transactional Memory. In *Procs of the 18th Intl Conf on Parallel Architectures and Compilation Techniques*, Sep 2009.

[95] Ravi Rajwar. *Speculation-Based Techniques for Transactional Lock-Free Execution of Lock-Based Programs*. PhD thesis, University of Wisconsin, Oct 2002.

[96] Ravi Rajwar and James R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Procs of the 10th Intl Symp on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[97] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *Procs of the 32nd Intl Symp on Computer Architecture*, Jun 2005.

[98] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. MetaTM/TxLinux: Transactional Memory for an Operating System. In *Procs of the 34th Intl Symp on Computer Architecture*, Jun 2007.

[99] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-Aware Transactional Memory for Increased Concurrency. In *Procs of the 41st Annual Intl Symp on Microarchitecture*, Nov 2008.

[100] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Procs of the 20th Intl Symp Distributed Computing*, Sep 2006.

[101] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based Transactional Memory with Scalable Time Bases. In *Procs of the 19th Symp on Parallelism in Algorithms and Architectures*, Jun 2007.

[102] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is Transactional Programming Actually Easier? In *Procs of the 15th Intl Symposium on Principles and Practice of Parallel Programming*, 2010.

[103] José M. García Tim Harris Adrián Cristal Osman Unsal Ibrahim Hur Mateo Valero Rubén Titos-Gil, Manuel E. Acacio. Hardware transactional memory with software-defined conflicts. In *Procs of the 5th Workshop on Transactional Computing*, Apr 2010.

[104] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a High Performance Software Transactional Memory System for a Multi-core Runtime. In *Procs of the 11th Intl Symp on Principles and Practice of Parallel Programming*, Mar 2006.

[105] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural Support for Software Transactional Memory. In *Procs of the 39th Annual Intl Symp on Microarchitecture*, Dec 2006.

[106] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing Signatures for Transactional Memory. In *Procs of the 40th Annual IEEE/ACM Intl Symp on Microarchitecture*, pages 123–133, Dec 2007.

[107] Sutirtha Sanyal, Adrián Cristal, Osman S. Unsal, Mateo Valero, and Sourav Roy. Dynamically Filtering Thread-Local Variables in Lazy-Lazy Hardware Transactional Memory. In *HPCC '09: Procs of the 11th Conf on High Performance Computing and Communications*, Jun 2009.

[108] William N. Scherer III and Michael L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Procs of the Workshop on Concurrency and Synchronization in Java Programs*, Jul 2004.

[109] William N. Scherer III and Michael L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Procs of the 24th Symp on Principles of Distributed Computing*, Jul 2005.

[110] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Procs of the 14th Symp on Principles of Distributed Computing*, Aug 1995.

[111] Arrvindh Shriraman and Sandhya Dwarkadas. Refereeing Conflicts in Hardware Transactional Memory. In *Procs of the 23rd Intl Conf on Supercomputing*, Jun 2009.

[112] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible Decoupled Transactional Memory Support. In *Procs of the 35th Intl Symp on Computer Architecture*, Jun 2008.

[113] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra Marathe, Sandhya Dwarkadas, and Michael L. Scott. An Integrated Hardware-Software Approach To Flexible Transactional Memory. In *Procs of the 34th Intl Symp on Computer Architecture*, Jun 2007.

[114] Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Procs. of the 20th Intl Symp on Parallelism in Algorithms and Architectures*, Jun 2008.

[115] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Procs of the 27th Intl Symposium on Computer Architecture*, Jun 2000.

[116] Janice M. Stone, Harold S. Stone, Phil Heidelberger, and John Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel and Distributed Technology*, 1(4), Nov 1993.

[117] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Procs. of the 14th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, 2009.

[118] Herb Sutter. The Trouble With Locks. *C/C++ Users Journal*, 23(3), Mar 2005.

[119] Michael M. Swift, Haris Volos, Neelam Goyal, Luke Yen, Mark D. Hill, and David A. Wood. OS Support for Virtualizing Hardware Transactional Memory. In *Procs of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, Feb 2008.

[120] Fuad Tabba, Andrew W. Hay, and James R. Goodman. Transactional Value Prediction. In *Procs of the 4th Workshop on Transactional Computing*, Feb 2009.

[121] J. Ruben Titos, Manuel E. Acacio, and Jose M. Garcia. Characterization of Conflicts in Log-Based Transactional Memory. In *Procs of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Feb 2008.

[122] Rubén Titos, Manuel E. Acacio, and Jose M. Garcia. Speculation-Based Conflict Resolution in Hardware Transactional Memory. In *Procs of the 23rd Intl Parallel and Distributed Processing Symposium*, May 2009.

[123] Rubén Titos-Gil, Anurag Negi, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. Zebra : A data-centric, hybrid-policy hardware transactional memory design. In *Procs od the 25th Intl Conf on Supercomputing*, Jun 2011.

[124] Sasa Tomic, Cristian Perfumo, Chinmay Kulkarni, Adria Armejach, Adrian Cristal, Osman Unsal, Tim Harris, and Mateo Valero. EazyHTM, Eager-Lazy Hardware Transactional Memory. In *Procs of the 42nd Intl Symp on Microarchitecture*, Dec 2009.

[125] Takayuki Usui, Yannis Smaragdakis, and Reimer Behrends. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *Procs of the 18th Intl Conf on Parallel Architectures and Compilation Techniques*, Sep 2009.

[126] M. M. Waliullah and Per Stenstrom. Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems. In *Procs of the 22nd Intl Conf on Parallel and Distributed Processing Symposium*, Apr 2008.

[127] David W. Wall. Limits of Instruction-Level Parallelism. In *Procs of the 4th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Apr 1991.

[128] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Procs of the 22nd Intl Symp on Computer Architecture*, Jun 1995.

[129] Tse-Yu Yeh and Yale N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Procs of the 25th Intl Symp on Computer Architecture*, Jun 1998.

[130] Luke Yen, Jayaram Bobba, Michael M. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Procs of the 13th Intl Symp on High-Performance Computer Architecture*, Feb 2007.

[131] Luke Yen, Stark C. Draper, and Mark D. Hill. Notary: Hardware Techniques to Enhance Signatures. In *Procs of the 41st Intl Symp on Microarchitecture*, Dec 2008.

[132] Lihang Zhao, Woojin Choi, and Jeff Draper. LagerTM: Cooperative Lazy-Eager Management for Improved Concurrency in Transactional Memory. In *Procs of the 20th Intl Conf on Parallel Architectures and Compiler Techniques*, Sep 2011.