Toward Lightweight and High-Performance Hardware Transactional Memory

Saša Tomić

Department of Computer Architecture

Universitat Politècnica de Catalunya

A dissertation submitted in partial fulfillment
of the requirements for the

Doctor of Philosophy in Computer Architecture

4th of June, 2011

Oficina de Doctorat
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Acta de calificación de tesis doctoral

Curso académico:

Nombre y apellidos

DNI / NIE / Pasaporte

Programa de doctorado

Unidad estructural responsable del programa

Resolución del Tribunal

Reunido el Tribunal designado a tal efecto, el doctorand / la doctoranda expone el tema de la su tesis doctoral titulada _____

_____.

Acabada la lectura y después de dar respuesta a las cuestiones formuladas por los miembros titulares del tribunal, éste otorga la calificación:

☐ APTA/O ☐ NO APTA/O

(Nombre, apellidos y firma) Presidente/a	(Nombre, apellidos y firma) Secretario/a	
(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal

_____, _____ de _____ de _____

El resultado del escrutinio de los votos emitidos por los miembros titulares del tribunal, efectuado por la Escuela de Doctorado, a instancia de la Comisión de Doctorado de la UPC, otorga la MENCIÓN CUM LAUDE:

☐ SI ☐ NO

(Nombre, apellidos y firma) Presidenta de la Comisió de Doctorado	(Nombre, apellidos y firma) Secretaria de la Comisión de Doctorado

Barcelona, _____ de _____ de _____

This dissertation is dedicated to my wife, Jasmina.
Without you, this would not have been possible.

Acknowledgements

I would like to acknowledge my advisors Mateo Valero, Adrián Cristal, and Osman Unsal for giving me the opportunity to attend PhD studies, and for their guidance, confidence, and patient demeanor later during the studies. They are the reason for my success on the professional plan, but also the reason for all the good time I had with my friends and my family in this beautiful city. Without Mateo's impressive ability to find new projects and fundings, my PhD studies would never happen.

I have been fortunate to be advised by Adrián Cristal and Osman Unsal. The last five years have been an incredible journey, during which I have benefited greatly from Adrián's and Osman's professional knowledge and leadership. In addition, Adrian and Osman are incredibly generous and supportive people, guiding me through both my professional and the personal quests.

My internship in Microsoft Research Cambridge was an unexpectedly enriching experience, that helped me to understand what research really is. I would like to thank Tim Harris for inviting me to the unforgettable three month internship and for providing me with his invaluable support and help, both during my stay in Cambridge and after I returned to Barcelona. Tim Harris showed me how hard work can and does pay off, how hard work provides huge satisfaction if you are building something useful, and how team work is the key for being successful.

I would also like to acknowledge all my great friends and colleagues from Barcelona Supercomputing Center (BSC) that supported me during my PhD studies and that shared with me their insight and their expertise. Many thanks go to Srđan Stipić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Ferad Zyulkyarov, Vladimir Subotić, Nehir Sonmez, Marco Galuzzi, Paul Carpenter, Nehir Sonmez, Ege Akpinar, Vladimir Marjanović, Vesna Smiljković, Vladimir Gajinov, Vasilis Karakostas, Nikola Marković, Otto Pflucker, Vladimir Čakarević, Petar Radojković, and many many others that I missed adding. I sincerely thank all you guys for your company during my PhD and for all the great time we had together.

My deepest thanks go to my wife, Jasmina. This dissertation would not have been possible without her love and support. She is both the source of my success and the reason that this success has meaning. I also thank Novak for the joy and happiness that he has brought to my life. Jasmina and Novak are my inspiration and my motivation, both in good times and bad times. They are, without any doubt, the best thing that ever happened to me.

My graduate work has been financially supported by the cooperation agreement between the BSC and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and by the European Commission FP7 project VELOX (216852).

Abstract

Conventional lock-based synchronization serializes accesses to critical sections guarded by the same lock. Using multiple locks brings the possibility of a deadlock or a livelock in the program, making parallel programming a difficult task. Transactional Memory (TM) is a promising paradigm for parallel programming, offering an alternative to lock-based synchronization. TM eliminates the risk of deadlocks and livelocks, while it provides the desirable semantics of Atomicity, Consistency, and Isolation of critical sections. TM speculatively executes a series of memory accesses as a single, atomic, transaction. The speculative changes of a transaction are kept private until the transaction commits. If a transaction can break the atomicity or cause a deadlock or livelock, the TM system aborts the transaction and rolls back the speculative changes.

To be effective, a TM implementation should provide high performance and scalability. While implementations of TM in pure software (STM) do not provide desirable performance, Hardware TM (HTM) implementations introduce much smaller overhead and have relatively good scalability, due to their better control of hardware resources. However, many HTM systems support only the transactions that fit limited hardware resources (for example, private caches), and fall back to software mechanisms if hardware limits are reached. These HTM systems, called best-effort HTMs, are not desirable since they force a programmer to think in terms of hardware limits, to use both HTM and STM, and to manage concurrent transactions in HTM and STM. In contrast with best-effort HTMs, unbounded HTM systems support overflowed transactions, that do not fit into private caches. Unbounded HTM systems often require complex protocols or expensive hardware mechanisms for conflict detection between overflowed transactions. In addition, an execution with overflowed transactions is often much slower than an execution that has only regular transactions. This is typically due to restrictive or approximative conflict management mechanism used for overflowed transactions.

In this thesis, we study hardware implementations of transactional memory, and make three main contributions. First, we improve the general performance of HTM systems by proposing a scalable protocol for conflict management. The protocol has precise conflict detection, in contrast with often-employed inexact Bloom-filter-based conflict detection, which often falsely report conflicts between transactions. Second, we propose a best-effort HTM that utilizes the new scalable conflict detection protocol, termed EazyHTM. EazyHTM allows parallel commits for all non-conflicting transactions, and generally simplifies transaction commits. Finally, we propose an unbounded HTM that extends and improves the initial protocol for conflict management, and we name it EcoTM. EcoTM features precise conflict detection, and it efficiently supports large as well as small and short transactions. The key idea of EcoTM is to leverage an observation that very few locations are actually conflicting, even if applications have high contention. In EcoTM, each core locally detects if a cache line is non-conflicting, and conflict detection mechanism is invoked only for the few potentially conflicting cache lines.

Contents

List of Figures

List of Tables

Listings

1 Introduction

After 50 years of exponential improvement in the performance of sequential execution, it becomes increasingly difficult to continue improving the performance of sequential processors [57]. By simply increasing the clock frequency, the dissipated energy becomes a serious issue, and the performance improvements are insufficient [47]. All computing vendors have now changed their strategy. Instead of improving the performance of single-core processors, they are increasing the number of processor cores, and thus started moving towards multi-core or many-core processors. Currently, vendor roadmaps promise the doubling of the number of cores per chip in the following years. These chips are variously called chip multiprocessors, multicore chips, and many-core chips. Sources as varied as Intel and Berkeley predict a hundred [37] if not a thousand cores [7] on a single chip.

Unfortunately, software developers were not prepared for the shift to multi-core processors. While improving the single-threaded programming methodologies for more than 50 years, they have done little on multi-threaded programming methodologies. Suddenly, software developers are faced with a challenge. They

have powerful processors, but they cannot use this processing power to improve the performance of their programs. They are not able to easily distribute the workload across all processor cores, to synchronize the calculations, and to collect the results. Current lock-based synchronization is very difficult to program, although it can provide good performance if a program is well written.

Transactional memory (TM) is an alternative synchronization method, which *promises* to help programmers write (1) efficient parallel programs with (2) safety and (3) ease. The underlying TM mechanism provides an Atomicity, Consistency, and Isolation (ACI) of a critical section, thus freeing the programmer from having to manually ensure them. At the moment, TM fulfils only a part of its promises. TM-based parallel programs are apparently easier to write and more often correct than lock-based parallel programs [58]. However, the performance and scalability of TM programs in not as good as of lock-based parallel programs.

TM is an optimistic concurrency control mechanism. TM is based on the idea of executing critical sections speculatively and atomically. If speculation was successful, all changes are made public, otherwise the changes are automatically undone, and the system returns to its state before the speculative execution started. Each execution of a critical section is called a *transaction*. The transaction are executed in parallel, while guaranteeing exactly-once semantics as if the transactions were run in a serial order. If conflicts are detected during the execution, some of these transactions are aborted to maintain consistency. When a transaction is aborted, all its speculative changes are reverted and the system is returned to the state before the transaction started execution.

In section 1.1, we outline the general challenge of synchronization in shared-memory parallel programs. Section 1.3 describes the promises of transactional memory and the challenges that this dissertation addresses.

1.1 Parallel programming challenge

After more than 40 years of parallel programming, writing parallel applications is still more difficult than writing sequential applications. It is more difficult to design, write, debug, and prove the correctness of a parallel algorithm than the equivalent sequential algorithm.

As Harris et al. [34] conclude, parallel programming is difficult because it lacks the support for abstraction and composition.

An *abstraction* is a simplified view of an entity, which captures the features that are essential to understand and manipulate it for a particular purpose. Abstraction hides irrelevant detail and complexity, and it allows humans (and computers) to focus on the aspects of a problem relevant to a specific task.

Composition is the ability to put together two entities to form a larger, more complex entity, which, in turn, is abstracted into a single, composite entity. Composition and abstraction are closely related since details of the underlying entities can be suppressed when manipulating the composite product.

Modern programming languages support powerful abstraction mechanisms, as well as rich libraries of abstractions for sequential programming. Procedures offer a way to encapsulate and name a sequence of operations. Abstract data types and objects offer a way to encapsulate and name data structures as well. Libraries, frameworks, and design patterns collect and organize reusable abstractions that are the building blocks of software. Stepping up a level of abstraction, complex software systems, such as operating systems, databases or middleware, provide the powerful, generally useful abstractions, such as virtual memory, file systems, or relational databases used by most software. These abstraction mechanisms and abstractions are fundamental to modern software development which increasingly builds and reuses software components, rather than writing them from scratch.

Parallel programming lacks comparable abstraction mechanisms. Low-level parallel programming models, such as threads and explicit synchronization, are unsuitable for constructing abstractions because explicit synchronization is not composable. A program component that contains explicit synchronization cannot be abstracted with black-box that has certain functionality. A program developer must be aware of the implementation details for the entire program, in order to avoid causing races or deadlocks.

1.2 Problems with parallel programming

Parallel applications are difficult to develop, due to various reasons. We name here two important reasons, that stand out from the rest. First, it is difficult to break

the functionality of a sequential program into equal-sized units of work, that can execute in parallel. Second, it is difficult to synchronize the parallel execution of the units of work, in a way that provides identical results as the original sequential execution.

Incorrect synchronization may result in unreliable execution because of:

- **Data races.** Parallel applications may have non-deterministic bugs, where an execution of parallel application produces results different from the equivalent sequential application.

- **Deadlocks.** It is very easy to lose track of locks and introduce deadlocks. For a deadlock, two or more threads create circular requests for locks. All threads in the circle block, and the execution of the deadlocked threads stops.

- **Livelocks.** In contrast with deadlocks, where the threads block, a thread here requests re-execution from other threads. No thread makes forward progress, while all threads in livelock perform work.

There are several levels of forward progress guarantees for parallel applications. The following three are commonly studied, and guarantee deadlock-free execution.

- **Wait-freedom** is the strongest of the three. It guarantees that a thread will make forward progress on *its own* work if the thread continues executing [38, 39]. This guarantee is very strong and, although desirable, typically results in poor overall performance.

- **Lock-freedom** guarantees that, if any given thread continues executing, then *some* thread will make forward progress with its work. Lock-freedom is sufficient guarantee for preventing livelocks in the system.

- Obstruction-freedom guarantees that a thread will be able to make progress with its own work, if other threads do not run concurrently (at the same time). Obstruction-free algorithms do not guarantee livelock-free execution.

Even a correctly synchronized parallel application may behave poorly, because:

- **Cache line bouncing.** Changing the same data from different processors causes cache-lines to bounce between private caches. This can limit the system throughput, as Larson et al. [46] observe.

- **Contention on critical sections.** Mutual exclusion guards of critical sections can needlessly restrict parallelism, by preventing non-conflicting accesses to critical sections. While finer-grain locks partly mitigate the problem, lock convoying and poorer cache performance can become a problem in this case, again reducing the performance.

A true solution for parallel programming must provide good performance (or a way to improve the performance), without risking the correctness of execution.

1.3 Transactional Memory for simpler parallel programming

Transactional Memory (TM) promises an elegant solution for many problems with parallel programming. With TM, a critical section is executed atomically, with an all-or-nothing semantics. That is, either a complete code of a critical section is executed, or none of it. No partial results of an execution may be visible at any moment. We call an execution of a critical section a "transaction".

The most important advantages of TM over locks are:

- **Improved parallelism.** A transactions may execute independently (in parallel) if it does not conflict with concurrent transactions. We say a transaction *conflicts* with concurrent transactions if it writes to locations that other concurrent transactions read or write. For example, read-only transactions can always execute in parallel. Read-write transactions are also non-conflicting if they access disjoint memory locations, or if they access memory locations at different time (not concurrently).

- **Deadlock and livelock freedom.** The underlying TM mechanism can ensure that the execution does not result in a deadlock or livelock. In this sense, TM mechanisms provide "lock-freedom" forward progress guarantee.

- **Composability.** Composition is the ability to put together two entities to form a larger, more complex entity, which, in turn, is abstracted into a single, composite entity. Achieving composability using locks requires from the programmer to get familiar with a specific implementation, introduce new locks and risk new deadlocks. In contrast, composing transactions [33] can be as simple as executing sub-transactions in the scope of a surrounding transaction.

A conflict between transactions can be *detected* either (1) during transactions execution, in which case we talk about *eager* conflict detection, or (2) when transactions commit, in which case we talk about *lazy* conflict detection.

After it detects a conflict between transactions, a TM mechanism performs a *conflict resolution* by, for example, aborting or stalling one of the conflicting transactions. Conflict resolution can again be either (1) eager, at the moment when a conflict is detected, or (2) lazy, at the moment it becomes necessary or appropriate.

1.3.1 Software vs. Hardware TM

Existing TM implementations are written purely in software (Software TM, or STM), with no particular hardware support or requirements, except compare-and-swap operations. This reflects in the performance of current TM implementations, since conflict detection requires intensive communication between software threads. The performance of STMs will improve in time, when STMs evolve and mature. However, it is unlikely that the performance of STMs will ever be close to the regular sequential execution, especially in high-performance programming languages like C or C++.

The evolution of STMs has already made a significant progress, from being completely unusable in practice, as analyzed by Cascaval et al. [21], to the current STM implementations which can actually reduce the execution time compared to the sequential execution, given enough processor cores [26]. The STM code is typically 4 or more times slower than the sequential execution. With 8 or more cores, the STM parallel application can execute faster than the original sequential application.

Without dedicated hardware support, the performance of single-threaded STM

executions will likely be much worse than the performance of the original sequential application. The appropriate question here is: which is the minimal hardware support that would provide the best TM performance and scalability? Pure-hardware TM (Hardware TM, or HTM) proposals, on the other end, have significantly lower overheads, with performance of single-threaded HTM executions very close to the performance of the original sequential application.

1.4 Dissertation Research Path

This dissertation brings us one step closer to a lightweight and high-performance TM alternative to locks. The proposed TM support should result multi-threaded synchronization mechanism that: (1) imposes minimal overhead to the execution, and (2) scales well with the number of cores. That is, the main goal of the presented TM support is to avoid any limits in parallel execution of independent transactions.

In this dissertation, I present several contributions I made to an already well-researched area of hardware support for TM. I see these contributions as being a competitive foundation for a real product, a processor with hardware support for TM that can be used ubiquitously instead for locks.

This is the research path I took towards writing this thesis:

1. **Choose a baseline HTM.** We first analyzed the high-level performance bottlenecks of HTMs, and tried to select a baseline HTM for my further research. I implemented an eager HTM, and another colleague from the group implemented a lazy HTM. According to our evaluations, lazy HTMs overall perform better from eager HTMs. The results of our evaluation align with the results of other researcher groups (for example [16]).

   The research we had gave us the knowledge that is very hard to extract from research papers – eager HTMs suffer from the same problems as lock-based synchronization mechanisms. Because of this, it is much more difficult to design an eager HTM protocol such that it efficiently works-around the associated problems (deadlocks, livelocks, etc.). It is even difficult to design an eager HTM that is obstruction-free (Section 1.2). The work-arounds in

eager HTMs become particularly visible when a workload has medium to high contention (rate of conflicts between transactions). These observations pushed us towards lazy HTMs.

2. **Simplify simulator development.** Analyzing several HTMs, together with their variations, required many changes in the simulators and often led to broken and non-working simulators. The simulators were sometimes breaking with all executions and sometimes only with particular executions. Finding bugs in simulators was very difficult and often required analyzing tens or hundreds of gigabytes of trace files – a task hardly possible for a human. The automation of the simulator testing functionality lead to a new methodology for continuous testing of simulators, which significantly simplifies and accelerates the simulator development.

3. **Analyze the performance of lazy HTMs.** After analyzing in details the functionality of lazy HTMs, it was obvious that its commit operation is very complex. The commit operation in lazy HTMs is composed of (1) transaction validation, and (2) publishing speculatively modified values. We observed that it is possible to decompose and simplify this complex operation, by (1) performing validation eagerly (during execution of transactions), and (2) resolving conflicts lazily (when necessary, at commit). This makes commits much simpler, faster, and with less overhead, and allows all non-conflicting transactions to commit completely in parallel. We name the resulting HTM an "**Ea**ger-La**zy**", or EazyHTM. We show more details on EazyHTM in Section 5.

4. **Eliminate other important problems with eager-lazy HTMs.** Other important disadvantage of lazy and eager-lazy HTMs is their lack of support for large transactions. EazyHTM stores all transactional data in private caches, and detects conflicts using metadata in private caches. If private caches are small, we have to fallback to software TM. This can be avoided by detecting conflicts in directory. This allows us to increase the transaction size to the size of memory directory, which is sufficient for current workloads. However, directly moving the conflict detection metadata from private caches to the

directory would require a significant amount of metadata (2 bits per processor core, per directory entry, or 64 bits for 32-core system). To address this, we came up with a mechanism for reducing the amount of metadata to the constant 2 additional bits per directory entry. The mechanism is based on classifying the lines between conflicting and non-conflicting. The classification is done automatically, in runtime. We discovered that conflicting lines are very uncommon, counted in tens of lines, while the non-conflicting are counted in hundreds and thousands. For the common non-conflicting lines we can use only 2 bit metadata. At the moment a line becomes conflicting, we can approach other mechanisms for conflict detection. The mechanism allows us to achieve the performance comparable with EazyHTM, with similar hardware cost, while supporting much larger transactions. We named this an **Eco**nomical HTM, or EcoTM, and we present it in more details in Section 6.

1.5 Contributions of this Dissertation

The most important contributions of this dissertations are the following:

- **A methodology for continuous testing of cycle-accurate simulators.** Cycle-accurate simulators frequently have bugs or incorrectly designed protocols. To accelerate the development, and to improve the reliability of simulators, we developed a methodology for dynamic runtime testing of cycle-accurate simulators (Section 4). The methodology dramatically reduced the testing and debugging time of the simulators, by continuously testing the simulators and, in case of an incorrect simulator execution, providing the exact moment a bug appears.

- **Separation of the conflict detection and resolution**, in order to simplify and accelerate transaction commit operation. While EazyHTM and EcoTM detect transactional conflicts eagerly, they defer the resolution of these conflicts until commit time. Besides having the advantage of knowing the conflicts during the transaction execution, the underlying system can be simpler,

as finding conflicts is not on the critical path of commit operation. Additionally, the whole mechanism is very flexible. For example, it is not necessary (although it is acceptable) to have split directories, and it is not necessary to use a specific interconnection topology.

- **A mechanism for allowing non-conflicting transactions to commit in parallel.** With EazyHTM and EcoTM, commits appear to be instantaneous to other running transactions. Furthermore, these improvements are provided while guaranteeing that pathological behavior [16] such as livelocks ("friendly fire"), starvation for writing transactions ("starving writer"), serialized commit or cascaded waits never occur.

- **A mechanism for efficiently reducing the transactional metadata storage requirements.** We propose the first HTM for large transactions that supports eager, lazy, and eager-lazy conflict management, providing efficiency and flexibility, while needing only 2 bits metadata per cache line. In comparison, a state-of-the-art HTM that supports only eager conflict management, TokenTM, requires 16 bits metadata per cache line.

- **An efficient unbounded, lazy-conflict-resolution HTM, that does not rely on Bloom-filter signatures.** Our conflict-detection protocol avoids false conflicts introduced in many other unbounded HTMs by Bloom-filter signatures. False conflicts in these signatures are much more likely with larger transactions, which we anticipate in future TM workloads. The evaluation indicates that our design has better performance with large transactions than the state-of-the-art HTMs.

2

Background on Transactional Memory

In this chapter, we introduce the concept of Transactional Memory (TM) in more details. We will first start with an introduction to the lock-based synchronization in Section 2.1, together with the problems with this kind of synchronization. In Section 2.2 we will discuss the main motivation for introducing TM-based synchronization as an alternative to locks. We will present the Software TM (STM) techniques, and then present the HTMs and their advantages over STMs.

2.1 Lock-based thread synchronization

Mutual-exclusion locks (or mutex locks, or simply locks) are one of the simplest and most commonly used thread synchronization constructs. Being a low-level synchronization mechanism, locks are (1) well performing, (2) flexible, and (3) complex to use correctly.

Physically, a lock is associated with a critical section of code. Lock allows only a single thread to enter the protected critical section. Ideally, each critical section

should have one associated lock. This lock needs to be *acquired* (or *locked*) by a thread before entering the critical section. Similarly, the thread needs to *release* (or *unlock*) the lock, before it leaves the critical section. If a lock is already acquired by another thread, a thread needs to wait until the lock is released. We call this lock contention.

Logically, a lock is associated with some shared data. A programmer has to keep track of logical and physical association of each lock, during entire development. To reduce the number of locks in the program, he can use one lock to synchronize access to multiple units of data. In this case we have larger *granularity* of locking, however, we might achieve less parallelism since we are more likely to have lock contention.

A bigger difficulty with locks are deadlocks. Deadlocks may occur if a thread holds some locks and tries to acquire a new lock. If it fails to acquire the new lock, the thread execution is suspended without releasing already acquired locks. This opens the door for a deadlock. For example, if a thread T1 acquires lock A and suspends while trying to acquire lock B, and thread T2 acquires lock B and suspends while trying to acquire lock A, we have a deadlock. Real-world scenarios of deadlocks can be far more complex.

As Herlihy et al. [39] define, a good lock-based synchronization algorithm should satisfy the following properties:

- **Mutual Exclusion.** This is a safety property and is clearly essential. To provide this property, the critical sections of different threads should not overlap. Each and all accesses to one shared data unit need to use the same lock.

- **Freedom from Deadlock.** The system should never "freeze". Individual threads may be stuck forever (*starve*), waiting on a lock, but some thread must make progress. To provide this property, a programmer should create an order between locks, and *always* acquire locks in the same order. Acquiring locks in the same order is error-prone. Unfortunately, an error can be unnoticed and appear non-deterministically, after program deployment.

- **Freedom from Starvation.** *Every* thread must eventually make progress. This property, while clearly desirable, is the least compelling of the three.

```
void do_transfer(int account1, int account2, int amount)
{
  int temp;
  temp = balance_get(account1);          // get current balance
  balance_set(account1, temp - amount); // withdraw money
  temp = balance_get(account2);          // get current balance
  balance_set(account2, temp + amount); // put money
}
```

Listing 2.1: Simplified bank-transfer example: sequential version

> There are some practical mutual exclusion algorithms that fail to be starvation free. These algorithms are usually deployed in circumstances where starvation is a theoretical possibility, but is unlikely to occur in practice. The starvation-freedom property is also weak in the sense that there is no guarantee for how long a thread waits before it enters the critical section.

A programmer needs to be consistent in lock usage. He needs to precisely define in which order, where, and for which data units is each lock acquired and released. Failure to do so may cause an application to block, without any obvious reason, and non-deterministically.

A particular complication arises when we manage (acquire and release) locks in different functions. Listing 2.1 presents a simplified sequential version example of a function that transfers money from account1 to account2. Current balance of an account is retrieved by calling a function "balance_get". A newly calculated amount of money is stored into the account by calling "balance_set".

Now, consider a lock-based parallel version, illustrated in Listing 2.2. It is difficult to know if the code will execute correctly. We need to know the details of the implementation of the functions *balance_get* and *balance_set*. Do they call some other functions? Are some locks acquired while they execute? Are deadlocks possible?

We cannot answer these questions without analyzing the complete code in details.

Developing correct and high-performance parallel programs using locks is a challenging task. The primary reasons are that (1) locks synchronize conservatively, and (2) while locks logically synchronize data accesses, they physically syn-

```
void do_transfer(int account1, int account2, int amount)
{
  acquire(lock_balance);
  int temp;
  temp = balance_get(account1);          // get current balance
  balance_set(account1, temp - amount); // withdraw money
  temp = balance_get(account2);          // get current balance
  balance_set(account2, temp + amount); // put money
  release(lock_balance);
}
```

Listing 2.2: Potentially incorrect lock-based parallel code for bank-transfer. If balance_get or balance_set acquire locks, a deadlock might arise.

chronize code accesses; the programmer has to maintain the correlation between the data and the code.

2.2 TM-based thread synchronization

Compared to the lock-based synchronization, transactional memory provides much simpler interface for synchronizing parallel threads. With TM, a programmer simply defines the borders of the critical section, and the TM engine makes sure that execution of that critical section does not result in any pathology commonly associated with lock-based synchronization. The focus of TM is not the performance, it is the elimination of the complexity associated with lock-based synchronization. TM eliminates the two primary reasons that make lock-based thread synchronization difficult – it eliminates the occurrence of deadlocks and simplifies the physical-logical association between the locks and the data. In addition, it adds some appealing properties to TM programs, such as composability of functions.

With TM synchronization, multiple threads can concurrently and speculatively enter the same critical section. A thread has an illusion that it is the only one entering the critical section at that moment. All the speculative updates of the thread are buffered in some private space and published if the speculation succeeds. We therefore say that a thread executes a critical section “in a transaction”. If the TM engine detects that the speculative execution of a transaction resulted in some anomaly (e.g., no equivalent serialization, a deadlock or a livelock), it rolls back

the changes and then restarts the affected transactions. All transactions are logically serialized in some order, although they may physically execute concurrently. The execution of a transaction appears to be indivisible and instantaneous to an external observer.

The origin of TM is in database theory. However, TM transactions have less properties than database transactions, since TM does not guarantee that the result of transaction execution will be visible after a program terminates. That is, TM transactions do not provide durability, which database transactions provide. The other properties of TM transactions are:

Atomicity requires that all constituent actions in a transaction complete successfully, or that none of these actions appear to start executing. It is not acceptable for a constituent action to fail and for the transaction to finish successfully. Nor is it acceptable for a failed action to leave behind evidence that it executed. A transaction that completes successfully commits and one that fails aborts.

Consistency is entirely application dependent, and it typically consists of a collection of invariants on data structures. If a transaction modifies the state of the world, then its changes should start from one consistent state and leave the data structures in another consistent state. Later transactions may have no knowledge of which transactions executed earlier, so it is unrealistic to expect them to execute properly if the invariants that they expect are not satisfied. Maintaining consistency is trivially satisfied if a transaction aborts, since it then does not perturb the consistent state that it started in.

Isolation requires that transactions do not interfere with each other while they are running – regardless of whether or not they are executing in parallel.

Relational Databases Management Systems (RDBMS) have been evolving for the last 50 years. Over time, they have been improving the performance while adding new functionalities. Some ideas that work very well in RDBMSs can also be applied to the TM systems. However, there are crucial differences between RDBMS and TM systems, and these differences prevent a direct application of the knowledge accumulated in RDBMS systems to TM:

- Traditional relational databases execute transactions for much longer time. The transactions in relational databases frequently make hard disk (permanent storage) accesses. Each hard disk access can last for millions of pro-

cessor cycles. In contrast, TM transactions cannot make I/O (that is, hard disk accesses), which makes them much shorter in time. Because of this, the speed of calculations affects the performance of TM transactions much more than database transactions. This has many direct consequences to the TM techniques.

A transaction in RDBMS can acquire its read and/or write when it starts the execution. If any conflicts occur with other RDBMS transactions, it is often more efficient in RDBMSs to make an order between transactions than to rollback some of them. If a suspended transaction becomes ready to commit, resuming it takes a relatively small percentage of time. In contrast, if a TM (and especially HTM) transaction should be suspended, it is instead typically more efficient to abort it and restart later, than to suspend it and resume it later.

- The RDBMS operations are given in a declarative SQL language, while TM instructions are typically from imperative languages. Declarative languages describe *what* the program should accomplish, instead of describing *how* should it be accomplished. This is in contrast with imperative programming, which requires an explicitly provided algorithm. In result, the same RDBMS query can typically be executed in a large number of ways, which can have a widely varying performance.

- Reads and writes in TM transactions are interleaved, where in RDBMS transactions discovering, reorganizing, and regrouping of reads and writes can take an insignificant percentage of the total execution time of a transaction. In RDBMS, the planning and optimizing a transaction execution can be much longer than in TM, without reducing the overall system performance. This allows various optimizations. For example, transaction commits can be made more common by moving all reads to the beginning of a transaction execution, and immediately taking locks for the writes. The data modifications can be performed at the end of transaction. This, and similar optimizations are not possible in TM transactions, since: (1) TM transactions have to finish their execution in much shorter time, and (2) some transactional accesses may be unknown until they actually happen, since some addresses may be

```
void do_transfer(int account1, int account2, int amount)
{
  tx_begin();
  int temp;
  temp = balance_get(account1);          // get current balance
  balance_set(account1, temp - amount); // withdraw money
  temp = balance_get(account2);          // get current balance
  balance_set(account2, temp + amount); // put money
  tx_commit();
}
```

Listing 2.3: Correct TM-based parallel code for bank-transfer

calculated on-the-fly, during transaction execution.

- TM transactions may have some non-transactional operations in parallel with transactional operations, while databases do not have the same problem.

TM-based synchronization provides much higher level of abstraction than lock-based synchronization. Using TM, the programmer does not need to indicate where to acquire or release locks, and does not have to worry about deadlocks or livelocks. The previous example of a bank transfer can be converted to a TM-version by making only trivial changes, shown in Listing 2.3.

We can distinguish three design choices in a TM implementation, which will be explained in more details in separate sections: conflict detection, conflict resolution, and version management.

2.2.1 Conflict detection

A TM system must mediate concurrent accesses to the same data. If some of the transactions that access data also modify it, we say that there is a conflict between transactions. For correctness, a TM system must detect and resolve all conflicts before the modifications are made public.

A conflict is detected when the underlying TM system determines that a conflict has occurred. Most HTMs use the cache coherence protocol to detect conflicts between transactions. A TM system may identify possible conflicts:

- Eagerly, during transaction execution [15, 17, 24, 55, 79], or

- Lazily, when a transaction tries to commit [19, 25, 30].

False conflicts. In HTM systems, for simplicity, conflicts are usually detected at the level of cache lines. If two transactions modify different parts of the same cache line, HTM systems typically mark a conflict between transactions, even though there is no actual conflict. We call this a false conflict. Another, much more important and common source of false conflicts is the approximative conflict detection, using some sort of signatures instead of the exact read and write set. A common example of such signature is Bloom-filter signature [12]. This is typically done to decouple the transactional subsystem from the rest of the processor. In this approach, a conflict is marked if a signature responds that a potential conflict with other transaction(s) exists [79]. However, the quality of Bloom-filter signatures is defined by the probability of "false positives", that is, a probability that a signature will claim that an entry has been added although it has not been.

There are many approaches to counter false conflicts. Reducing false conflicts may improve performance to some extent, since we reduce the number of unnecessary aborts. However, in order to reduce the number of false conflicts, we may have to introduce other performance overheads, or increase the hardware requirements. To counter false conflicts that are due to false sharing of a cache-line, we may use smaller granularity of conflict detection (word instead of cache line) [31], or value-based instead of address based conflict detection [59, 75]. To reduce the number of false conflicts in Bloom-filter signatures we can use larger signatures or use better hash functions [61, 65, 80]. However, both approaches are only a partial solution, that is, they only partly reduce the probability of false conflicts.

2.2.2 Conflict resolution

A TM system needs to resolve a detected conflict, in order to ensure isolation between concurrent transactions. Conflicts between transactions can be resolved either:

- Eagerly, as soon as they are detected [15, 17, 24, 55, 79], or
- Lazily, when a transactions is ready to commit [19, 25, 30].

A common eager resolution policy is to stall conflicting transactions and, if stalling introduces a risk of a deadlock, then to abort some of the conflicting transactions. Both stalling and aborting have their advantages and disadvantages. Stalling may result in less wasted work than directly aborting a transaction, but it risks deadlocks. On the other hand, aborting may result in more wasted work, but does not introduce deadlocks.

Conflicts need to be resolved if they impact the correctness, or if they prevent *serializability* of transactions. Serializability states that the result of a concurrent transaction execution must be identical to a result in which these transactions are executed in some serial order. Serializability simplifies reasoning of an execution, since it allows a programmer to write a transaction in isolation, as if no other transactions is executing in the system concurrently. The TM system is free to reorder, or to interleave transactions, but it must ensure the result of their execution remains serializable. Although serializability requires that transactions appear to run in some sequential order, the transactions may actually run in parallel, as long as the assumed sequential order is respected.

2.2.3 Version management

A TM implementation needs to track and manage transaction updates, that is, the tentative work of a transaction. If a transaction aborts, the version management mechanism needs to revert the changes made during transaction execution. In our example, a TM implementation needs to track the updates to the balances of $account1$ and $account2$.

TM systems typically manage updates by either using:

- Eager versioning, in which case the memory is directly updated, as soon as possible, and a copy of the original values is created [15, 17, 55, 79], or
- Lazy versioning, in which case the updates are buffered until a transaction commits successfully [19, 25, 30].

Initial HTM proposals were keeping speculatively accessed lines in private caches. If these lines could not fit into private caches, a transaction was aborted. More re-

cent HTM proposals also store speculative lines in: (1) load-store queue, (2) higher levels of the memory hierarchy, or (3) software-managed storage.

Most HTM are either monolithically eager or monolithically lazy. That is, they either perform conflict detection, resolution, and version management all eagerly, during transaction execution, or they delay all these operations until transaction tries to commit. We will now describe the operation of purely or monolithically eager and lazy HTMs.

2.3 Eager TM

Eager TM is also known as pessimistic concurrency-control TM, or an pessimistic TM.

Eager TM performs conflict detection, resolution, and version management together, on each instruction execution. When a transaction is about to access a location, an eager TM system: (1) detects a conflict, (2) resolves it, and (3) updates the memory location. This type of concurrency control allows a transaction to claim exclusive ownership of data prior to proceeding, preventing other transactions from accessing it.

In result, a transaction has very simple commit operation, since there is very little work to do for commit [15, 17, 55, 79]. This type of TM targets workloads with very low probability of conflicts.

2.4 Lazy TM

Lazy TM is also known as an optimistic concurrency-control TM, or an optimistic TM.

A Lazy TM detects and resolve a conflict lazily, when a transaction tries to commit. Lazy concurrency control allows multiple transactions to access and modify the same data concurrently, and to continue running even if they conflict. However, the TM must detect and resolve all conflicts before a transaction wants to commit.

In result, a transaction in lazy TM needs to perform significant work when it

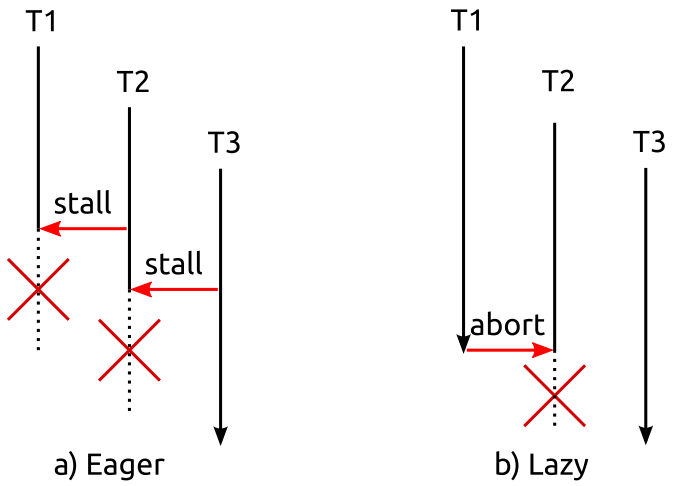


Figure 2.1: Why lazy conflict resolution performs better under contention?

wants to commit. However, the overall system performance can be higher than in eager TM, since the transactions do not have to communicate during their execution (in order to resolve conflicts) until some of them wants to commit.

2.5 Lazy versus eager TM

To see how conflict management can affect the performance of a TM system, consider the example shown in Figure 2.1, inspired by the work of Spear et al. [73].

Eager conflict detection (Figure 2.1a) attempts to minimize the amount of wasted work performed in the system. Here, transaction T1 conflicts with T2, and is stalled. After this, T2 conflicts with T3, and gets stalled too. Note that though T1 does not conflict with T3, it must stall until T3 (and then T2) either aborts or commits. Most eager HTM implementations suffer from these so-called *cascading waits* [16].

With lazy conflict detection (Figure 2.1b), all transactions execute until a transaction attempts to commit. In the example, when T1 attempts to commit, it only aborts T2. Once T2 aborts, T3 can also commit without conflicts.

In the above example, we make two observations. First, lazy conflict resolution allows two transactions to commit, while eager resolution allows only one. This difference is due to a fundamental facet of eager conflict resolution: it must address

potential conflicts (caused by an offending access to a shared location), while lazy resolution deals with conflicts that are *unavoidable* in order to allow a transaction to commit. Second, attempts by eager systems to reduce "wasted" work are not always successful. In Figure 2.1a, for instance, the eager system stalls T1. Since T1 does not eventually abort, the work that was avoided had not been wasteful. Even if T1 did abort, the amount to work saved would be minimal, due to the small size of transactions.

In summary, eager HTM systems can suffer from the following problems:

1. Must speculate which transaction is more likely to commit (and which should be aborted) when an offending access is attempted. At this time, the system has little information, but needs to speculate, which is inherently suboptimal. Solving this problem accurately is a complex problem algorithms similar to the one presented by Smith et al. [70].

2. Even if they make a successful prediction, a chain of waiting transactions still cause a cascading wait and the system needs to avoid deadlocks that may arise out of such (cascaded) stalls. Alternatively, if conflicts result in aborts, performance degrades due to unnecessary aborts.

Bobba et al. [16] and Shiraman et al. [68] illustrate how lazy conflict resolution can allow more parallelism than eager conflict resolution, especially under high-contention workloads. In high-contention workloads the contention management mechanism frequently has to make a difficult decision of which is the best transaction to abort or stall. Making a good decision in eager systems can require complex algorithms, which translate to complex hardware. In contrast, lazy systems have a simple policy that generally works well: the committer wins. Beside guaranteeing forward progress (since some transaction has to commit), the policy avoids deadlocks and livelocks. Furthermore, lazy conflict detection can result in higher performance than eager conflict resolution if the amount of "wasted work" in lazy systems is offset by the "wasted time" in eager systems.

2.6 Mixing the transactional and non-transactional accesses

Transactional memory execution may have non-transactional accesses together with transactional accesses. The TM implementation may behave differently when transactional and regular code try to access the data of another transaction. Blundell et al. [13] explored the problem and described the two common behaviors of TM systems.

Weak isolation (also known as “weak atomicity”) guarantees transactional semantics only among transactions. Non-transactional accesses to transactional data may create various types of problems. For example: (1) unambiguous data races between transactional and non-transactional code, (2) granularity problems that occur when the data managed by the TM implementation is coarser than the program variables being accessed, (3) accesses by aborted transactions in TMs that do not order transaction aborts, or (4) when the programmer attempts to use transactional accesses to one piece of data to control whether or not another piece of data is shared.

Strong isolation (also known as “strong atomicity”) guarantees transactional semantics not only between transactions, but with non-transactional code as well. Providing strong isolation with software transactional memory can greatly affect the overheads of the TM implementation [2, 66]. To provide strong isolation, HTMs have natural advantage over STMs. Since HTMs operate at a low level of instruction execution, they can track both transactional and non-transactional accesses without significant additional complexity.

3

Evaluation Environment

The best kind of evaluation for hardware proposals would be to measure the performance using real-world applications, on a real-world operating system and other commonly present factors of the typical execution environment. Unfortunately, with the current hardware technology it is not possible to modify the processors, and we have to approach alternative evaluation methods.

To simulate a processor, we use a modern and stable architectural simulator, M5, from the Advanced Computer Architecture Laboratory of the University of Michigan [11]. The simulator code is split into several functional modules and provides a flexible basis for simulating different computer system architectures. It features pervasive object-oriented orientation, with major simulation structures (CPUs, busses, caches, etc.) represented as objects, both externally and internally.

M5 simulator directly supports two interchangeable CPU models: (1) a simple, functional, one-CPI CPU, and (2) a detailed model of an out-of-order SMT-capable CPU. Both models use a common high-level ISA description. Other models could also be created, if it becomes necessary. In this research, we used the CPU model

with the simple one-CPI CPU.

M5 simulator supports full-system simulation, and system-call emulation. In full-system simulation, it executes a real Operating System from a disk image. In system-call emulation, M5 emulates the functionality of the Operating System, and the time spent in the OS functions is typically ignored. In this work, we used the full-system version of M5 and the Linux kernel version 2.6.18.

The M5 simulator features a detailed, event-driven memory system including non-blocking caches and split-transaction buses. These components can be arranged flexibly, e.g., to model complex multi-level cache hierarchies.

The M5 simulator decouples ISA semantics from its timing CPU models. This enables support for multiple ISAs, and M5 currently supports: Alpha, ARM, SPARC, MIPS, POWER and x86 ISAs.

The M5 simulator runs on most operating systems (Linux, MacOS X, Solaris, OpenBSD, Cygwin) and architectures (x86, x86-64, SPARC, Alpha, and PPC). It is readily portable to other hosts and other Unix-like operating systems that are supported by GCC. Alpha binaries to run on M5 (including the full Linux kernel) can be built on x86 systems using gcc-based cross-compilation tools, so no Alpha hardware is needed to make full use of M5.

M5 simulator provides full-system simulation support for:

- Alpha: The M5 simulator models a DEC Tsunami system in sufficient detail to boot unmodified Linux 2.4/2.6, FreeBSD, or L4Ka::Pistachio.
- ARM: The M5 simulator can model up to four cores of a Realview ARM development board with sufficient detail to boot unmodified Linux 2.6.35+ with a simple or out-of-order CPU.
- SPARC: The M5 simulator models a single core of a UltraSPARC T1 processor with sufficient detail to boot Solaris in a similar manner as the Sun T1 Architecture simulator tools (building the hypervisor with specific defines and using the HSMID virtual disk driver).
- x86: The M5 simulator supports a standard PC platform.

We started using M5 while its support was only stable for Alpha architecture. Therefore, all our evaluations are based on DEC Alpha Tsunami system. Since

M5 simulator matured in the meantime, it should be possible to port our implementations to other architectures without any major difficulties.

Unmodified M5 simulator supports a snooping bus-based coherence protocol for modeling symmetric multiprocessor (SMP) systems. Because a complete system is just a collection of objects (CPUs, caches, memory, etc.), multiple systems can be instantiated within a single simulation process.

For the purpose of this research, we extended the original M5 cache-coherence from a simple bus-based to a more elaborate directory-based cache coherence. We have also implemented a 2D mesh core-to-core interconnection network (ICN). The simulator has a sequential consistency memory model.

The HTM modules for EazyHTM and EcoTM connect with the rest of the M5 simulator using clear and strictly defined interface. This allows using different HTM implementations with almost no changes in the rest of the simulator. The LogTM-SE and Scalable-TCC HTMs, which our group developed in order to compare their performance with EazyHTM and EcoTM, all use virtually identical base simulator.

Selecting an appropriate evaluation workload is of particular importance. Since we simulate processor, the program execution takes much more time. Our simulator provides around 2 Million Instructions Per Second (MIPS) on a high-end server. This speed is very high compared to other architectural simulators, but the simulator is much slower than real processors. For example, Intel 486DX provides around 54 MIPS, and recent desktop machines provide more than 10,000 MIPS.

3.1 STAMP benchmark suite

To provide TM researchers with larger transactions, Cao Minh et al. [20] proposed a *Stanford Transactional Applications for Multi-Processing* (STAMP). STAMP quickly became a *de facto* standard for evaluating TM proposals. STAMP is written in C, and it can execute workloads with various TM back-ends, software, hardware, or hybrid TM. The TM back-ends can be selected and configured by changing TM macros in one header file.

STAMP strives to be a TM benchmark suite with: (1) breadth – having a variety of algorithms and application domains, (2) depth – covering a wide range of ap-

Application	Domain	Description
Bayes	machine learning	Learns structure of a Bayesian network
Genome	bioinformatics	Performs gene sequencing
Intruder	security	Detects network intrusions
Kmeans	data mining	Implements K-means clustering
Labyrinth	engineering	Routes paths in maze
SSCA2	scientific	Creates efficient graph representation
Vacation	online transaction processing	Emulates travel reservation system
Yada	scientific	Refines a Delaunay mesh

Table 3.1: The applications in the STAMP suite

plication behaviour: high and low contention, short and long transactions, small and large transactions, and (3) portability – supporting software, hardware and hybrid TMs. Table 3.1 summarizes applications from the suite, and the Table 3.2 summarizes their characteristics: transaction length, read and write set size, and the contention.

We will now present a summary of each application from the suite.

Bayes. This application implements an algorithm for learning the structure of Bayesian networks from observed data. The main data structure is adtree, used to estimate the probability distributions. The Bayesian network itself is represented as a directed acyclic graph, with a node for each variable and an edge for each conditional dependence between variables. On each iteration, each thread is given a variable to analyze, and as more dependencies are added to the network, connected subgraphs of dependent variables are formed.

A transaction is used to protect: (1) the calculation and (2) addition of a new dependency. The calculation result depends on the extent of the subgraph that contains the variable being analyzed. Bayes spends almost all its execution time in long transactions with large read and write sets. This benchmark has a high amount of contention as the subgraphs change frequently.

Genome. Genome assembly is the process of taking a large number of DNA segments and matching them to reconstruct the original source genome. This program has two phases to accomplish this task: (1) creating a hash-set of unique segments, and (2) each thread tries to remove a segment from a global pool of unmatched segments and add it to its partition of currently matched segments.

Application	Read set min	Read set average	Read set max	Write set min	Write set average	Write set max	Transaction Length	Transaction Time	Contention
Bayes	1	203.9	3377	0	91.9	2578	Long	High	High
Genome	1	35.6	198	0	4.8	42	Medium	High	Low
Intruder	0	25.2	58	0	6.9	23	Short	Medium	High
Kmeans	2	6.4	8	0	1.7	2	Short	Low	Low
Labyrinth	4	590.5	893	1	367.3	476	Long	High	High
SSCA2	1	3.99	4	0	1.99	2	Short	Low	Low
Vacation	23	75.3	102	1	9.8	18	Medium	High	Low/Medium
Yada	1	82.7	457	0	29.7	192	Long	High	Medium

Table 3.2: The characteristics of the evaluated applications from the STAMP TM benchmark suite, with the number of cache lines of read and write sets, evaluated with 16 simulated processors. While many applications have less than 50 cache lines in the write set, bayes for example, at some point of the execution has more than 2500 lines.

Transactions are used in each phase of the benchmark. The transactions are of moderate length and have moderate read and write set sizes. Almost all of the execution time is transactional, and there is little contention.

Intruder. Signature-based network intrusion detection systems (NIDS) scan network packets for matches against a known set of intrusion signatures. Network packets are processed in parallel and go through three phases: capture, reassembly, and detection. The main data structures are: (1) a simple FIFO queue for the capture phase, and (2) dictionary (self-balancing tree) in the reassembly phase. The dictionary contains lists of packets that belong to the same session.

Transactions are used in the capture and reassembly phases. This benchmark has relatively short transactions. It also has moderate to high levels of contention depending on how often the reassembly phase rebalances its tree. Overall, since two of the three phases are spent in transactions, this benchmark has a moderate amount of total transactional execution time.

KMeans. The K-means algorithm groups objects in an N-dimensional space into K clusters. This algorithm is commonly used to partition data items into related subsets. Each thread processes a partition of the objects iteratively and inside a transaction. The amount of contention among threads depends on the value of K, with larger values resulting in less frequent conflicts as it is less likely that two threads are concurrently operating on the same cluster center.

The contention is low, since threads only occasionally update the same center concurrently. The transactions are small, since their size is proportional to D, the dimensionality of the space. Overall, the majority of execution time for KMeans is spent calculating the new cluster centers, in a non-transactional code.

Labyrinth. This benchmark routes paths in a maze. The main data structure is a three-dimensional uniform grid that represents the maze. In the parallel version, each thread grabs a start and end point that it must connect by a path of adjacent maze grid points. The calculation of the path and its addition to the global maze grid are enclosed by a single transaction. A conflict occurs when two threads pick paths that overlap. Each transaction initially creates a private copy of the grid, and uses this copy for calculating a path. To add the path to the global grid, the transaction revalidates by re-reading all the grid points along the new path. If validation fails, the transaction aborts and the process is repeated, starting with a

new, updated copy of the global grid.

Creating the private grid copy adds the entire grid to the read set of the transaction. To reduce the probability of conflicts, a TM needs to early-release the global grid. Early-release allows a transaction to remove a data address from its transactional read set so that it does not generate conflicts. However, the programmer or compiler must guarantee that removing the address from the read set does not violate the atomicity of the program. Path calculation takes almost all execution time, and this creates very long transactions with very large read and write sets. Virtually all of the code is executed transactionally, and the amount of contention is very high because of the large number of transactional accesses to memory.

SSCA2. Scalable Synthetic Compact Applications 2 (SSCA2) is comprised of four kernels that operate on a large, directed, weighted multi-graph. These four graph kernels are commonly used in applications ranging from computational biology to security. For STAMP, we focus on Kernel 1, which constructs an efficient graph data structure using adjacency arrays and auxiliary arrays. This part of the code is well suited for TM as it benefits greatly from optimistic concurrency. The transactional version of SSCA2 has threads adding nodes to the graph in parallel and uses transactions to protect accesses to the adjacency arrays. Since this operation is relatively small, not much time is spent in transactions. Additionally, the length of the transactions and the sizes of their read and write sets is also small. The amount of contention is the application is relatively low as the large number of graph nodes leads to infrequent concurrent updates of the same adjacency list.

Vacation. This application implements an online transaction processing system, serving the task of emulating a travel reservation system. The system is implemented as a set of trees that keep track of customers and their reservations for various travel items. During the execution of the workload, several client threads perform a number of sessions that interact with the travel system's database. In particular, there are three distinct types of sessions: reservations, cancellations, and updates.

Each of these client sessions is enclosed in a coarse-grain transaction to ensure validity of the database. Consequently, vacation spends a lot of time in transactions and its transactions are of medium length with moderate read and write set sizes. Low to moderate levels of contention among threads can be created by increasing

the fraction of sessions that modify large portions of the database. Finally, using transactions greatly simplified the parallelization as designing an efficient locking strategy for all the data structures in vacation is non-trivial.

Yada. Yet Another Delaunay Application (YADA) benchmark implements Delaunay mesh refinement. The basic data structures are: (1) a graph that stores all the mesh triangles, (2) a set that contains the mesh boundary segments, and (3) a task queue that holds the triangles that need to be refined. The goal of the algorithm is to produce a mesh without skinny triangles, that is, maximizing the minimum angle of all the angles of the triangles in the triangulation. In each iteration of the algorithm, a skinny triangle is removed from the work queue, its retriangulation is performed on the mesh, and any new skinny triangles that result from the retriangulation are added to the work queue.

Transactions enclose accesses to the work queue, as well as the entire refinement of a skinny triangle. As almost all the execution time is spent calculating the retriangulation of a skinny triangle, this benchmark has relatively long transactions and spends almost all of its execution time in transactions. While performing the retriangulation, several triangles in the mesh are visited and later modified, leading to large read and write sets and a moderate amount of contention.

In our evaluations we have used the parameters proposed for HTM systems, presented in Table 3.3 on Page 33.

Application	Arguments	Description
Bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2 -t NUMPROC	Dependencies for v variables are learned from r records, which have $n \times p$ parents per variable on average. Edge insertion has a penalty of i, and up to e edges are learned per variable.
Genome	-g256 -s16 -n16384 -t NUMPROC	Gene segments of s nucleotides are sampled from a gene with g nucleotides. A total of n segments are analyzed to reconstruct the original gene.
Intruder	-a10 -l4 -n2048 -s1 -t NUMPROC	n traffic flows are analyzed, a of which have attacks injected. Each flow has a max of l packets, and the random seed s is used.
KMeans-Hi	-m15 -n15 -t0.05 -i random2048-d16-c16.txt -p NUMPROC	The number of cluster centers used is varied from m to n. A convergence threshold of t is used, and analysis is performed on input i. The input consists of n points of d dimensions generated about c centers.
KMeans-Low	-m40 -n40 -t0.05 -i random2048-d16-c16.txt -p NUMPROC	
Labyrinth	-i random-x32-y32-z3-n96.txt -t NUMPROC	The input i consists of a maze of dimensions $x \times y \times z$. n paths are routed.
Vacation-Hi	-n4 -q60 -u90 -r16384 -t4096 -c NUMPROC	The database has r records of each reservation item, and clients perform t sessions. Of these sessions, u% reserve or cancel items and the remainder create or destroy items. Sessions operate on up to n items and on q% records.
Vacation-Low	-n2 -q90 -u98 -r16384 -t4096 -c NUMPROC	
Yada	-a20 -i 633.2 -t NUMPROC	The input mesh i is refined so that it has a minimum angle of a. The input 633.2 consists of 1264 elements.

Table 3.3: STAMP parameters used in our evaluation

4

Dynamic Runtime Testing for Error-Free Cycle-Accurate Simulators

4.1 Introduction

The proposals for hardware changes are typically first implemented and evaluated on architectural cycle-accurate simulators. These simulators aim to accurately represent the functionality, the interaction, and the timing of all functional components of the real hardware. As such, architectural simulators are typically very complex and prone to errors. A simulator with errors can unnecessarily delay the evaluations of architectural proposals. Incorrect simulator evaluations can take future product development in a wrong direction, or create other unnecessary development costs. Simulator developers often invest significant effort in thoroughly testing and verifying the simulators, attempting to confront the errors.

Verification and debugging are often seen as the most difficult problems in today's complex hardware and software systems. This is especially the case with

the products that require continuous modifications. It is commonly estimated by many hardware and software companies that verification will take between 50 and 70 percent of the total cost of a product [36, 43]. For large or mission-critical projects, verification can take as much as 90 percent of the total cost. Traditional testing methods (for example, unit testing [63]) require a significant amount of programming effort to provide good confidence in simulator correctness. However, architectural simulators are often changed rapidly and extensively, used to evaluate a certain idea or approach, and after that the changes are discarded. Thus, the testing of architectural simulators is often performed irregularly and unsystematically.

In contrast with simulators, architectural emulators (for example, QEMU [9]) model far fewer details of the target hardware architecture. The functionality of the emulators typically consists only of: (1) decoding instructions, (2) executing them, and (3) updating the simulated memory. The objective of an emulator is to provide a functional equivalent of the target architecture, without estimating its performance. Emulators are typically used to make virtual machines and to do cross-platform software development. Since emulators are far simpler than simulators, they are generally much more stable, much easier to debug, and to validate. Still, executions on an architectural simulator and an emulator have to produce identical final results.

This work presents *dynamic runtime testing*, a development methodology that verifies the functional correctness of a cycle-accurate simulator during its entire development cycle. Dynamic runtime testing discovers the unintentional functional errors (bugs) in a simulator by comparing its execution with an execution of the integrated simple emulator. The emulator serves as a golden-reference for a functional verification of the simulator. In dynamic runtime testing, we execute both the simulator and the emulator sequentially and in the same environment. We compare their execution as often as possible, preferably after every operation, and any difference in the executions of the simulator and the emulator indicates a possible bug in the simulator and needs to be carefully examined. Dynamic runtime testing aims to be a “write and forget” methodology for continuous testing, where developer creates the testing environment and then continues to freely change the simulator. The developer can be relaxed, knowing that the simulator will report

any bugs, even during rapid simulator prototyping.

The functional correctness of the simulator is dynamically verified during the entire execution of a simulator, in every simulator execution, and during the entire lifetime of a simulator. In Section 4.2, we explain a procedure for applying dynamic testing to almost any architectural simulator, either to the simulator as a whole, or to a specific component (module) of the simulator. Then we show several use cases of the methodology: coherent multi-level caches, Hardware Transactional Memory (HTM), and Out-Of-Order (OOO) processors.

Dynamic runtime testing can detect only functional bugs. This is a trade-off between the effort needed for implementing the technique and the achieved functionality. To detect other types of bugs, we still have to complement dynamic testing with other testing methods. Dynamic runtime testing can be complemented with a variety of testing and debugging techniques. In Section 4.3 we explain our motivation for detecting only functional bugs, and we mention some other techniques that we used for testing the simulators.

In Section 4.4, we explain how a developer can use dynamic runtime testing to find and fix bugs in a simulator. We describe our preferred debugging methods – execution tracing and an interactive debugger tool. We also present an example of a debugging session of a simulator that has dynamic runtime testing. If it detects a potential bug, dynamic testing provides a direct path for finding the bug, and for verifying that the bug has been eliminated. We show a simple and efficient procedure that can help to locate the section of code with a bug. The procedure is much faster and has much less room for errors than a typical debugging procedure.

In Section 4.5, we evaluate the impact of dynamic runtime testing on the performance of two cycle-accurate simulators: coherent multi-level caches and Hardware Transactional Memory (HTM). The overhead of dynamic testing is modest (10-20%) in our implementations, since the baseline simulators are much more complex than the simple emulators added for dynamic runtime testing. The overhead of dynamic testing could be even smaller in other implementations, for example, if we test a full-system cycle-accurate simulator of a pipelined out-of-order architectural processor. In this case we can use a highly optimized architectural emulator, which can provide speed close to the native execution [9]. In contrast, the fastest full-system cycle-accurate simulators can simulate only around 2 MIPS

(million instructions per second). Even if we assume that an architectural emulator induces a 10 times slowdown, this is more than 1000 MIPS on a modern machine, which is about 500 times faster than the 2 MIPS of the complex architectural simulator. Thus, the overhead of such a configuration could be less than 1%.

In Section 4.6, we share our experiences with dynamic testing. Dynamic testing helped us to rapidly develop, test, and verify several architectural cycle-accurate simulators. Consequently, our simulator development became more productive and more efficient. In particular, dynamic testing provides us the following advantages over other simulator testing methods:

1. Faster simulator testing, since we do not need to create a complex and extensive test suite,

2. Faster simulator debugging, since we can pinpoint a precise moment and the circumstances that lead to a bug, instead of only discovering that a bug appeared, and

3. Faster simulator development, since we have more confidence and freedom to develop the simulator, knowing that any introduced bug will immediately appear.

In addition, dynamic runtime testing could help to recover the simulator from a certain type of bugs. If simulator execution is different from the emulator, it is possible to fallback to the execution results of the emulator. This can improve the overall reliability of the simulator, although admittedly not its correctness.

4.2 Detecting Bugs Using Dynamic Testing

In this section, we present the dynamic simulator testing methodology. We start with a high-level overview of the methodology and then present several use cases, simultaneously showing more details on the implementation of the methodology.

Dynamic testing can be applied both to individual components of a simulator (examples in Sections 4.2.1 and 4.2.2), or to the entire simulator (example in Section 4.2.3). In the further text, we will use a generic term “simulator” even for

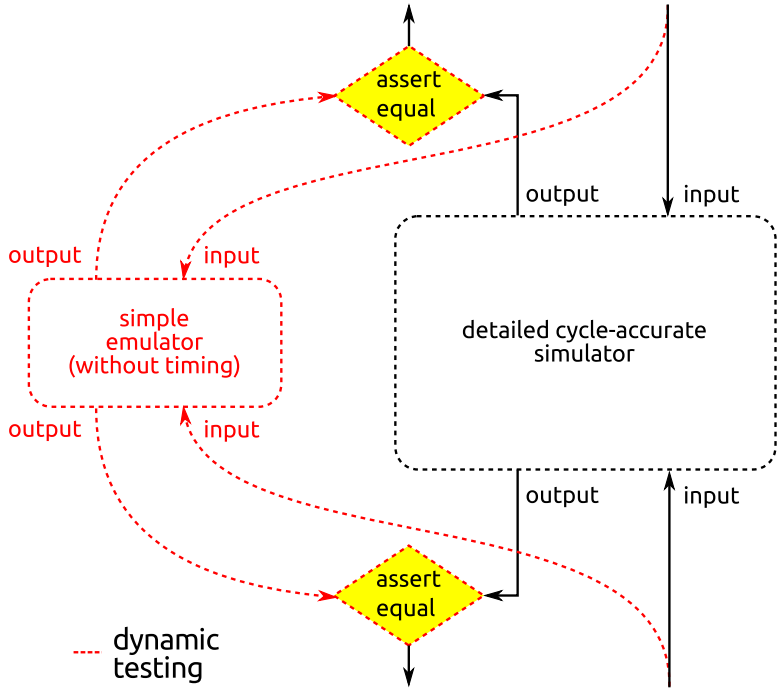


Figure 4.1: An overview of dynamic testing. The tested simulator (black) and the functionally identical emulator (red) have to be produced the same output during entire simulator execution. Any difference indicates a likely bug.

individual simulator components, since the individual simulator components can usually be transformed to independent simulators.

An overview of dynamic runtime testing is illustrated in Figure 4.1. Dynamic testing consists of comparing (1) the outputs of a functional simulator, with (2) the outputs of its functionally equivalent *emulator*. The comparison is done after every executed operation, and all outputs have to be identical. Although any type of output could be compared, we found it sufficient to compare the values of memory locations.

A high-level overview of the procedure for implementing dynamic testing can be represented as:

1. **Emulator integration.** We make a functionally-equivalent emulator and integrate its code with the baseline simulator. The emulator should not provide any timing estimations, and it should focus on being simple, well performing, and functionally correct.

2. **Emulator validation.** We disable the code of the baseline simulator and redirect its input (e.g., operations and memory values) to the emulator. We have to confirm that all applications terminate correctly and do not give any errors or warnings.

3. **Simulator-emulator comparison.** Finally, we re-enable the code of the baseline simulator giving it the same input as to the emulator. We execute an operation in the simulator, after that in the emulator, and then compare the outputs. Any difference in the outputs of the simulator and the emulator indicates a possible bug in either the simulator or the emulator.

Although we did not do so, it is possible to execute the simulator and the emulator in parallel (multi-threaded), and to synchronize their execution in order to verify the correctness. In our view, the added complexity of synchronization would not compensate for the added value of potentially faster execution. In that approach, we would check and synchronize the progress of the simulator and the emulator after each executed operation, get the results (outputs) of the two executions, and compare them. The overhead of synchronization can easily exceed the overheads

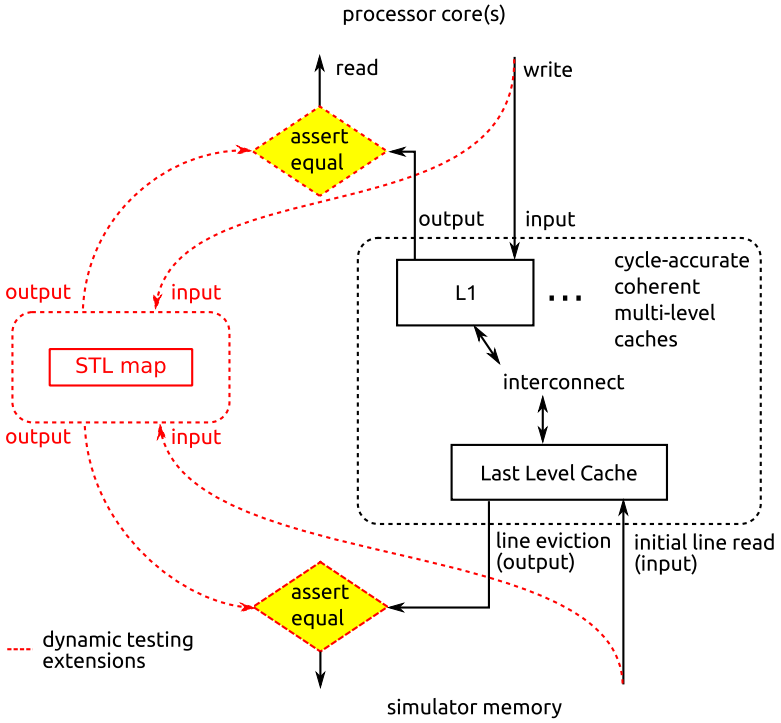


Figure 4.2: Dynamic runtime testing applied to coherent multi-level caches. The cache lines fetched and evicted by the (1) the cache emulator (STL map) and (2) the cycle-accurate coherent caches, must have the same value.

of the sequential execution of the emulator, especially in the case when the execution of the emulator is short.

The simulator notifies a developer, and provides an exact point of execution at which the difference from the emulator appeared. In case there is no difference between the outputs between the simulator and the emulator, we can be highly confident that the simulator-based evaluations are functionally correct, but still not certain. Dynamic simulator testing cannot guarantee that no bugs have remained in the simulator. However, assuming that the simulator executes a wide set of applications, the majority of bugs are likely to be discovered.

In the following sections, we demonstrate dynamic testing with several real-world use cases.

4.2.1 Use Case: Coherent Multi-level Caches

Coherent multi-level caches are functionally simple, although their implementation can be very complex. Our cycle-accurate simulator for the coherent multi-level caches is a collection of objects (one object per cache structure) that: (1) uses a coherence protocol and state machines to track the ownership of cache lines, (2) tracks the values (data) of the cache lines, and (3) calculates the access latency of each access.

Bugs in coherent multi-level caches usually appear in the coherence protocol, which can lead to multiple "modified" copies of the same location at different instances or levels of cache, resulting to incorrect values of some locations. Our goal was to eliminate the frequently-buggy coherence protocol and to avoid multiple copies of cache lines. This can be achieved with a cache emulator that has only one level and that is directly accessible by any part of the simulator. Such emulator obviously cannot estimate an access latency, but this is not the objective of the emulator.

A single level of caches allows us to further simplify the code. By analyzing the requirements, we can conclude that the same functionality can be provided by a generic data container for key-value pairs. The data container stores the pairs of (1) an address of a cache line and (2) the data stored in the cache line. Beside the data container, we wrote simple functions for extracting sequences of bytes from a cache line. Most modern programming languages provide such data containers, typically with a name *map*, or *a dictionary*. For example, C++ has a Standard Template Library (STL) map, which supports adding a new key-value pair, updating the value stored at a certain key, and removing some or all entries.

Dynamic runtime testing checks the following functionalities of multi level caches: (1) every read from a location needs to return the last value written by any processor to the same location, and (2) every write-back from the caches to the simulator memory needs to return the last written value. These functionalities must be satisfied at all times, by all types of coherent caches: bus-based, directory-based, broadcast-based or other, with any cache-interconnection topology and interconnection type.

Figure 4.2 illustrates the resulting configuration of coherent multi-level caches

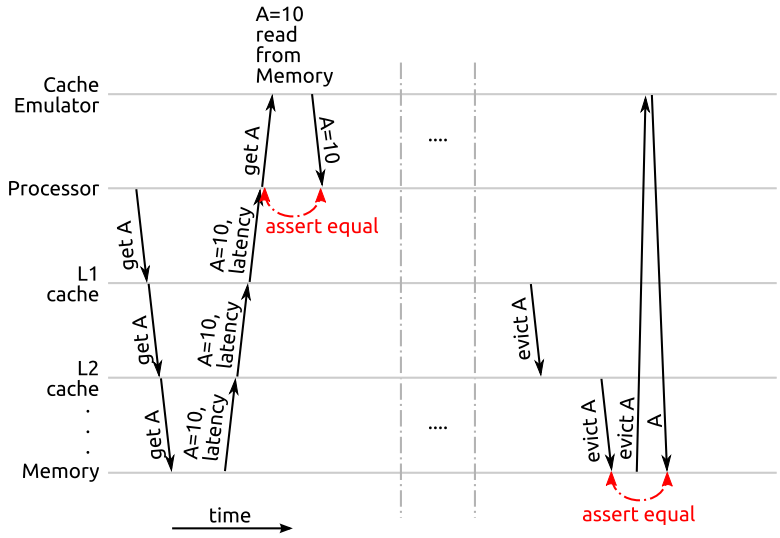


Figure 4.3: The time diagram of dynamic testing for the coherent multi-level caches. The cache-simulator and the cache-emulator execute sequentially, and have to return the same values.

READ

```
// cycle-accurate cache simulator. Multi-level
data = processor[cpuid]->L1->read_data(address, size, &latency);

// cache emulator. Single-level
data_functional = functional_cache.read_data(address, size);
if (data_functional != data) {
   FATAL_ERROR("incorrect cache value: %x instead of %x", data, data_functional);
   // also print the simulator cycle, state, and the accessed address, and then exit
}
```

WRITE

```
// cycle-accurate cache simulator. Multi-level
processor[cpuid]->L1->write_data(address, size, data, &latency);

// cache emulator. Single-level
functional_cache.write_data(address, size, data);
```

LINE EVICT (from the last-level-cache, LLC)

```
// LLC evicts the line with address "address" and data "data"
data_functional = functional_cache.read_data(address, size);
if (data_functional != data) {
   FATAL_ERROR("incorrect data in evicted line address: %x", address);
   // also print the simulator cycle, state, and the accessed address, and then exit
}
```

Figure 4.4: Pseudocode of dynamic runtime testing for the coherent multi-level caches

that includes dynamic runtime testing. When program reads the a value, the processor requests the value from the multi-level cache simulator, which may have to fetch the value from the main memory of the simulator, since the main memory always has all cache lines. The objects in the cache simulator communicate by exchanging messages, and each communication between cache objects increments the total latency of a cache access. In the end, when the processor receives the value from the multi-level cache simulator, it also receives the estimated latency of the access, and uses this latency to schedule the execution of the thread.

Figure 4.3 presents a time diagram of the dynamic testing of caches. When a processor requests a value from its L1 cache, the request may propagate to L2 cache or higher memory levels. After the request is completed, and the cache simulator returns the value and the latency of the access, the processor gets the value of the same location from the cache emulator. A code in the processor then confirms that the two values (from the simulator and the emulator) are the same. The same process is performed by all processors in the system, and with all their cache accesses. When a location is evicted from the top-level cache, the same location is also evicted from the cache emulator, and the code in the simulator memory confirms that the two evicted values are the same.

In Figure 4.4, we show the pseudo-code of our implementation of dynamic testing for the cache simulator. A read returns the requested value and checks that the value is the same in both the simulator and the emulator. A write updates the values in two caches without doing any checks. If the cache simulator needs to evict a line, the same location is also removed from the cache emulator, and the data in the two cache lines are checked to be identical. If the data is identical, it is stored in the simulated main memory. Otherwise, the difference is reported to the developer since it indicates a probable bug in the implementation of the coherent multi-level caches. Having the exact point of the execution where the difference appeared, the debugging of the cache-coherence protocol is much simpler.

4.2.2 Use Case: Hardware Transactional Memory

In our past work, we implemented and evaluated several proposals of Hardware Transactional Memory (HTM). Transactional Memory [34] is an optimistic con-

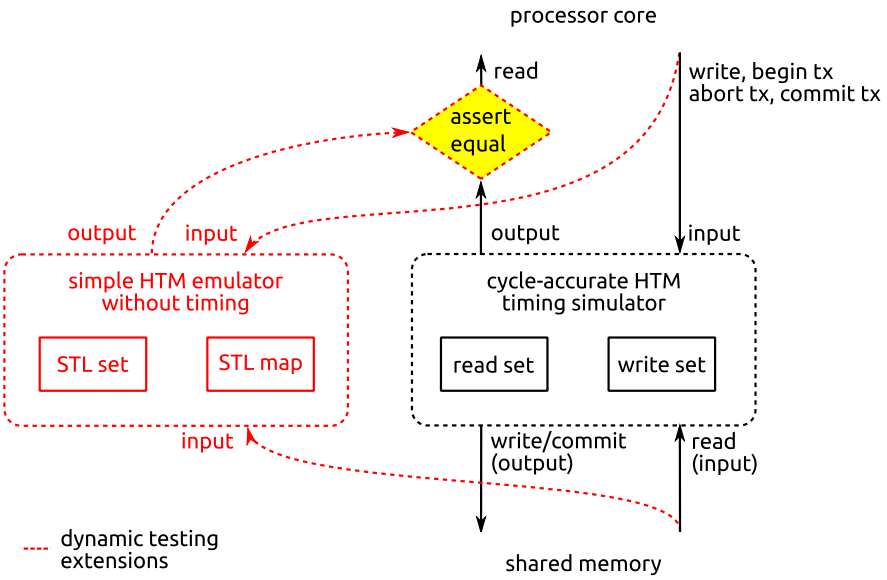


Figure 4.5: Dynamic runtime testing applied to HTMs. All reads are compared between the HTM simulator and the HTM emulator, and must return the same value. Optionally, writes/commits could be compared as well.

currency mechanism, which allows different threads to execute speculatively the same critical section, in a "transaction". The assumption is that the speculative execution of the transaction will not write over the data used by other concurrent transactions. In case the assumption was correct, we say that the speculation is successful, the transaction "commits" and publishes the speculative writes made during its execution. Otherwise, we say that a transaction has a conflict with some other transaction(s), and the HTM system decides which of the conflicting transactions are aborted. If a transaction is aborted, the speculative writes made by this transaction are rolled back, and the execution of this transaction is restarted.

The actual HTM protocol for publishing and rolling back speculative writes can be very complex, often leading to bugs in commits and aborts of transactions. To improve performance, a designer of an HTM protocol may decide to partially clear the transactional metadata during transaction commit [55], or to group-change the permissions of all speculatively written lines [30].

To design a reference HTM emulator, we tried to eliminate complex commit and abort procedures, providing only the basic functionalities universal to all HTMs.

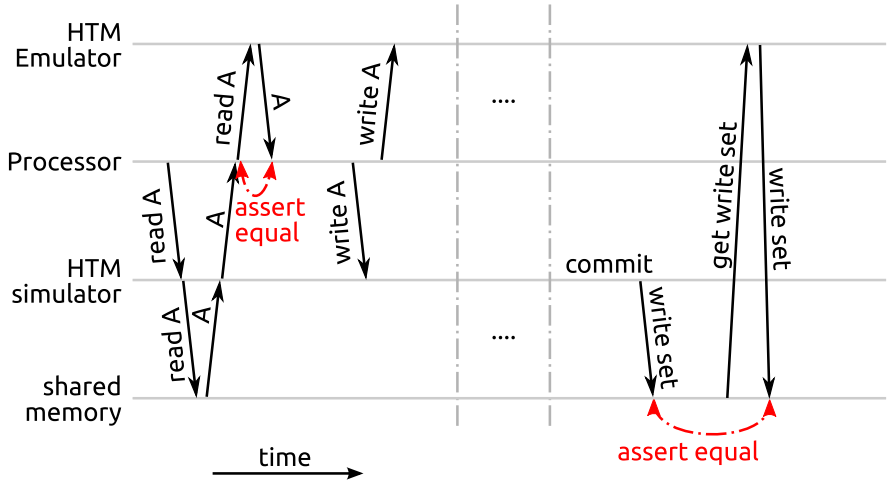


Figure 4.6: The time diagram of dynamic testing of an HTM. The HTM-simulator and the HTM-emulator execute sequentially, and have to return the same values.

The *first* necessary functionality of an HTM emulator is to buffer the speculative writes until a transaction successfully commits. We can keep the speculatively writes in an STL map (similar to the cache emulator). The *second* necessary functionality of an HTM emulator is the detection of conflicts with other transactions. A transaction needs to check the speculative reads and the writes with all other active transactions. Since we already track the speculative writes in the STL map, we only need to track the speculative reads in another STL set. Since STL map and set have theoretically unlimited capacity, the reference HTM emulator can also successfully detect the problems usually caused by limited hardware resources in HTMs.

Figure 4.5 shows a graphical overview of the presented approach for dynamic testing of an HTM. The same HTM emulator can be used to test HTMs with eager and lazy version management and can verify the values of both speculative reads and writes. In case a transaction already speculatively wrote to the location, a read from the same transaction has to return this speculatively written value. Otherwise, a read has to return the last non-speculative value of the location in the system. A transaction has to commit all values speculatively written during its execution, and it has to commit the last written values of these locations.

Figure 4.6 presents a time diagram of dynamic runtime testing of an HTM. During execution, a simulated processor sends the memory accesses and the transactional events, first to an HTM simulator and after that to the HTM emulator. The

```
TRANSACTIONAL READ

data                =                HTM.write_set[txid].get(addr) or caches.get(addr)
data_functional = FunctionalHTM.write_set[txid].get(addr) or shr_mem.get(addr)
if (data_functional != data) {
   FATAL_ERROR("incorrect HTM value: %x instead of %x", data, data_functional);
   // also print the simulator cycle, state, and the accessed address, and then exit
}

TRANSACTIONAL WRITE

if not HTM.write_set[txid].has(addr):
   HTM.write_set[txid].fetch_from_caches(addr)
HTM.write_set[txid].update(addr) with data

if not FunctionalHTM.write_set[txid].has(addr):
   FunctionalHTM.write_set[txid].fetch_from_shr_mem(addr)
FunctionalHTM.write_set[txid].update(addr) with data

ABORT

FunctionalHTM.abort_instantly(txid)  // instantly clears the write_set & restarts
HTM.initiate_abort(txid)             // rollback & restart; may take many cycles

COMMIT

FunctionalHTM.abort_conflicting_transactions(txid) // detect & resolve conflicts
FunctionalHTM.commit_to_shr_mem(txid)  // instantly publishes specul. changes

// regular HTM: starts conflict detection/resolution/committing specul. changes
// this may take many cycles
HTM.initiate_commit(txid)
```

Figure 4.7: Pseudocode of the implementation of dynamic runtime testing for an HTM

values that transactions read, and the committed values, are compared between the two HTMs. Any difference from the HTM emulator indicates a likely bug in the HTM simulator. The simulator logs the difference together with more details on the simulator state (for example, simulator clock). Based on the log, a developer can start debugging precisely at simulator state where the potential bug appeared.

Figure 4.7 shows the pseudo-code of our implementation of dynamic testing for HTMs. To simplify the code of the HTM emulator and at the same time make it less dependent on the particular implementation of the HTM simulator, we decided to slightly relax our implementations of dynamic testing of HTMs. Our implementations do not verify the committed values. Instead, a transaction in a single cycles publishes all its speculative writes, by updating the values in the cache emulator (the STL map) described in Section 4.2.1. On the other side, the cycle-accurate HTM simulator publishes the speculatively writes by interacting with the multi-level cache simulator, in a process that may take many cycles, and may require many changes of the permissions of the cache lines.

4.2.3 Use Case: Out-of-Order Simulator

Dynamic testing can also be applied to an entire cycle-accurate Out-Of-Order (OOO) processor simulator. The biggest problem with OOO processor simulators are their hard-to-find bugs which appear only with certain values or certain interleaving of instructions, which may appear only in very long simulations. Many bugs are related to incorrect implementations of some instructions or their parts (micro-operations). These bugs may eventually cause some memory location to have incorrect values, which may change the execution after millions or billions of instructions, making debugging almost impossible.

Dynamic runtime testing can significantly improve the stability of OOO simulators since it detects these bugs instantly, as they happen. In Figure 4.8, we present a schematic overview of a possible implementation of dynamic runtime testing for OOO simulators. This use case is slightly different from previous examples of dynamic testing, since a reference emulator has only one input (from simulator memory) and one output (to simulator memory). However, this does not significantly change the implementation of dynamic runtime testing, compared to the previous

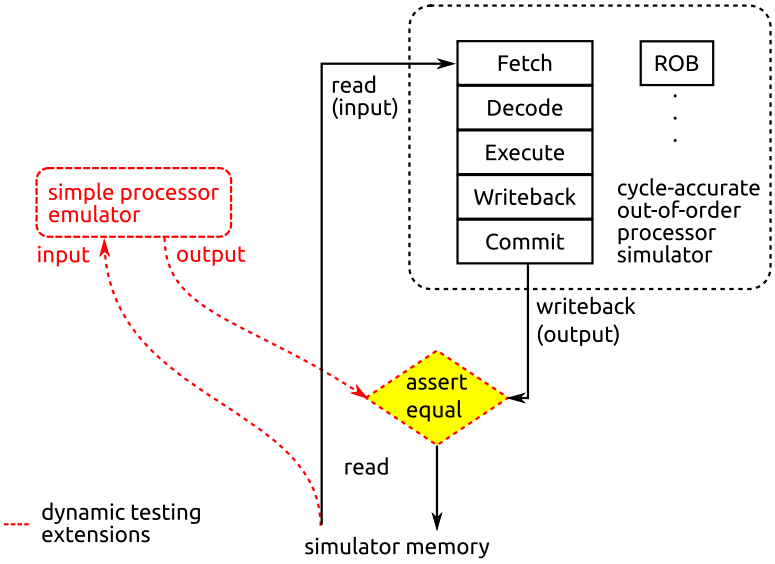


Figure 4.8: Dynamic runtime testing applied to the entire Out-of-Order simulator.

examples. To dynamically test an OOO simulator, we can compare its writes to the simulator memory with the writes made by a simple processor emulator. Having identical memory writes during the *entire* simulation provides a strong confidence that the cycle-accurate OOO processor simulator is functionally correct.

Since processor emulators are much faster than cycle-accurate OOO simulators (two or more orders of magnitude), the dynamic testing should not significantly affect the speed of the simulator. Cycle-accurate OOO simulators are inherently slower from the emulators since they simulate the functionality and the interaction of all hardware elements physically present in OOO processors, while processor emulators typically only decode instructions, execute them, and then update the simulated memory.

4.2.4 Other Use Cases

Similarly to the given examples of dynamic testing, the same principle could be used to improve the functional correctness of other cycle-accurate hardware simulators, and to simplify their debugging without significantly reducing their performance. In general, a tested hardware simulator should evaluate an extension or a modification representable by a simple, functional emulator. Among other

examples, dynamic runtime testing can be used for: single-processor multi-level memory hierarchy, incoherent multi-level memory hierarchy, system-on-chip simulators, network models, on-chip routing protocols, or pipelined processors.

4.3 Non-functional Bugs in a Simulator

Dynamic Runtime Testing only guarantees that a simulator completed the simulation without any functional errors. That is, the presented development methodology cannot *guarantee* that a simulator will give a correct or accurate estimation of the execution time, although it can help to eliminate many bugs. An example of an error that dynamic runtime testing does not see is an incorrect calculation of the execution time (as long as the execution is otherwise correct).

While it is possible to extend dynamic runtime testing with a basic testing of various other estimations provided by a simulator, we decided not to do that. Instead, we aimed for the simplicity of the methodology instead of generality.

As Dijkstra commented, testing shows the presence, not the absence of bugs. Therefore, to have higher confidence in our evaluations, we have given our best in making an extensive set of tests for correct evaluations of execution time. To test and validate our timing estimations, we have used the following methods in addition to dynamic runtime testing: manual testing, code review, unit testing, regression testing, and asserting invariants during execution. These testing methods are described elsewhere in the literature, for example in [3].

4.4 Finding and Fixing Simulator Bugs

After dynamic testing reports a potential bug in the simulator, a developer needs to approach the conventional debugging methods in order to find the source of the bug in the simulator. We describe here two common debugging methods: (1) a conventional debugging tool, or a debugger, for example gdb [74], and (2) execution traces.

Debugger allows a developer to stop the execution of the simulator at the moment he finds the most appropriate, and to examine the state of the simulator

memory and the architectural registers. This allows the developer to examine in details the complete state of the simulator, and even it even allows him to test the output of particular simulator functions, or to manually set the values of some memory locations. Debuggers generally have good performance and support for advancing the execution "forward in time". Unfortunately, going "back in time" is very difficult in a debugger. This means that if a developer misses the point of failure, he generally has to stop the simulator execution, restart the simulator, and then wait until the execution comes to the same point.

Trace-based debugging does not require a specific tool, since it consists of instrumenting the simulator code, for print the important part of simulator context to a trace file. Having a static trace file allows a developer to explore the execution not only by advancing "forward in time" (as with typical debuggers), but also "back in time", with no added complexity. By analyzing the static trace file, a developer can reason about the state of the simulator and expect that a bug appeared in a certain section of simulator code.

However, in certain aspects, trace-based debugging may be more complicated from a debugger. First, a developer needs to instrument the code for tracing, while developing the simulator. If the trace files do not contain all the information that a developer needs, he needs to re-instrument the simulator code, make more verbose trace files, and to re-execute the complete simulation. He similarly needs to remove some tracing instrumentation if the trace files are too verbose, which makes them unreadable and unnecessarily large.

Our approach to a trace-based debugging is to *turn off* tracing *by default*. This improves the execution speed of a simulation and reduces the storage requirements. In essence, this eliminates the trace files for all executions without bugs. We enable tracing after dynamic runtime testing reports a potential bug. Our implementation of tracing also has several levels of verbosity. While more verbose trace files provide more information on the simulator states, they slow down the execution more, and are slower or more difficult to analyze later.

```
# execute the simulator evaluation
$ ./build/ALPHA_FS/m5.fast ./configs/example/fs.py ...

# the execution status of the simulator
...
simulation clock 3.068 MIPS
simulation clock 2.967 MIPS
simulation clock 3.163 MIPS
simulation clock 3.159 MIPS
simulation clock 3.143 MIPS
simulation clock 3.133 MIPS
simulation clock 3.111 MIPS
simulation clock 3.074 MIPS
m5.fast: cpu/simple/atomic.cc:612: Fault AtomicSimpleCPU::read(uint64_t, T&, unsigned int) [with T = uint64_t]:
Assertion `data_correct == data64' failed.
Program aborted at cycle 2101189739000
...
```

Figure 4.9: An example of Dynamic Testing: the simulator reports a potential bug

4.4.1 An Example of a Debugging Session

In this section, we show an example of how dynamic runtime testing can simplify simulator debugging.

In our simulator development, we prefer using the trace-based debugging and we use a debugger only if necessary. Trace-based debugging provides an easy way to analyze the execution of the simulator both forward and back in time, starting from any position in the simulator execution.

If dynamic runtime testing detects a possible bug, it reports the bug on the "standard error" stream, and then stops the execution of the simulator. We show an output of the described execution scenario in Figure 4.9.

After we turn on tracing, we re-run the execution that uncovered a bug. Dynamic runtime testing now generates not only the address and the value of the location with the incorrect value, but also a complete trace of all memory accesses (addresses and values of reads and writes) that preceded the bug. From our experience, a bug is most often created in the last operation performed over the location. Less frequently a bug is 2-3 operations before, and rarely earlier than that. To find the previous uses of the location that has an incorrect value, we analyze the traces using standard text-processing tools, for example, *grep*, *sed*, and *awk*.

Figure 4.10 shows an example of a trace file. The last line in the trace file holds the address of the location with an incorrect value. In this particular case, the address of the variable is 0xfb4b5c8, and the address of the cache line holding the variable is 0xfb4b5c0.

In the next step, we "grep" the trace file to find the most recent occurrences

```
# check the last lines of the execution trace
$ tail htm_trace.txt
T0*   TXRD   line 0xf9619c0: 0xf9619e0=0x0   OWL
T0*   TXRD   line 0xf9619c0: 0xf9619e8=0x0   OWL
T0*   TXRD   line 0xf9619c0: 0xf9619f0=0x0   OWL
T0*   TXRD   line 0xf9619c0: 0xf9619f8=0x0   OWL
T0*   TXRD   line 0xf961a00: 0xf961a00=0x0   OWL
T0*   TXRD   line 0xf961a00: 0xf961a08=0x0   OWL
T0*   TXRD   line 0xfb4b5c0: 0xfb4b5c8=0x0   OWL
*** ERROR: P0   RD     line 0xfb4b5c0: 0xfb4b5c8=0x120043220 and should be 0x0
```

Figure 4.10: An Example of Dynamic Testing: potential bug found in the log

```
# check the last usage of this line
$ grep 0xfb4b5c0 htm_trace.txt | tail

T0    TXRD   line 0xfb4b5c0: 0xfb4b5c0=0x0   ---
T0    TXRD   line 0xfb4b5c0: accb->txnoconfl=0, accb->txwr=0
T0    TXRD   line 0xfb4b5c0: 0xfb4b5c8=0x0   --L
T0    TXRD   line 0xfb4b5c0: 0xfb4b5c8=0x0   --L
T0    TXWR   line 0xfb4b5c0: 0xfb4b5c8=0x0   --L
T0    0xfb4b5c0 TXWR accb->txnoconfl=1, accb->txwr=0
T0     Line 0xfb4b5c0 evicted, moving to overflow buffer!
T0*   TXRD   line 0xfb4b5c0: 0xfb4b5c0=0x0   OW-
T0*   TXRD   line 0xfb4b5c0 searching in shadow space
T0    shadow ??? --> orig 0xfb4b5c0 no overflow buffer entry
T0*   TXRD   line 0xfb4b5c0: 0xfb4b5c8=0x0   OWL
*** ERROR: P0   RD     line 0xfb4b5c0: 0xfb4b5c8=0x120043220 and should be 0x0
```

And here is our problem!
The value should be in the overflow buffer, but the overflow buffer does not have it!

Figure 4.11: An Example of Dynamic Testing: a potential cause of the bug found. The overflow buffer in lazy HTM does not have the value that it should have.

of our cache line. The filtered trace for the cache line is shown in Figure 4.11. Reading the final operations over the cache line, we see the following. The cache line 0xfb4b5c0 had to be evicted from the L1 cache, and moved to an "overflow buffer" (sort of a victim cache for transactional data). However, the next time we accessed the overflow buffer, our cache line was not in it. This means that the bug could be in the code for moving the value to the overflow buffer, or in the code for the retrieving a value from the overflow buffer.

We now have an exact segment of simulator code that has a bug, and we can see the interleaving of accesses that lead to the incorrect behavior. After we analyze the functionality of the simulator code with a bug, we can very quickly identify and fix the problem.

A problem may arise if the trace files do not hold enough information. For example, imagine that in our case we did not log the operations with the overflow buffer. In this case, we have to increase the verbosity of trace files and to repeat the simulator executions. Verbose tracing provides more details on the simulator state during execution, and this often results in easier debugging. Since all simulator

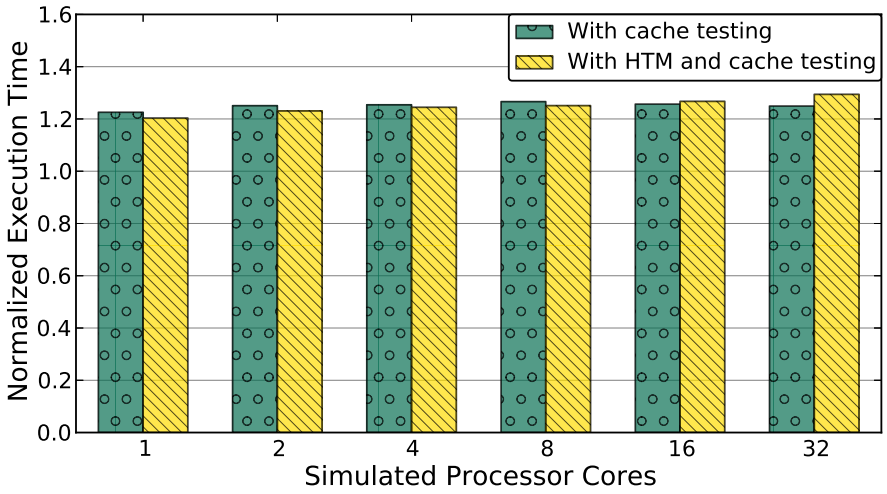


Figure 4.12: Dynamic testing impact to the simulator speed during Operating System (OS) booting. The average simulator speed is normalized to the one without dynamic testing.

executions are deterministic, changing the verbosity of traces and re-executing the simulator will produce the same bug, even if we execute a multi-threaded application inside the simulator.

In this example, finding and fixing a bug was easy. In some other cases, a bug can be more difficult to find and we have to use a debugger, or some other debugging method. In all cases, it is very important to re-execute the complete benchmark suite after we verify that a bug has been eliminated in a single benchmark configuration, since fixing one bug might uncover or create other bugs in different benchmark configurations.

4.5 Evaluation

In this section, we evaluate the performance impact of dynamic testing on simulator performance (execution time). We have used the M5 full-system simulator [11] as a base architectural simulator, and extended it to implemented MESI-directory coherent multi-level caches, and several HTM proposals.

We carried out all simulations on modern Intel Xeon X86_64 processors, taking care of minimizing the I/O and other system calls, which may non-deterministically affect the simulator performance. As a result, all simulator executions have more

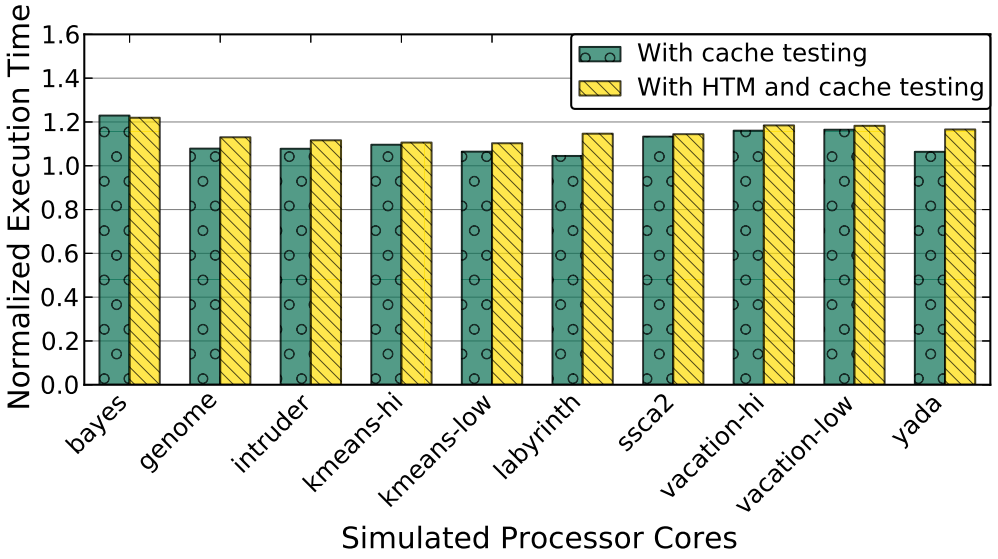


Figure 4.13: Dynamic testing impact to the simulator speed during application execution. The average simulator speed is normalized to the one without dynamic testing.

than 98% CPU utilization on average. We have measured the execution time of the simulator for all applications from the STAMP transactional benchmark suite [20], and for 1, 2, 4, 8, 16, and 32 simulated processor cores. The simulator is single-threaded, and to simulate multi-core processors, the simulator sequentially processes events of each simulated processor core or device. We have repeated each execution three times to reduce the effect of wrong measurements in single executions caused by random, uncontrollable events, and then calculated an average execution time.

Figure 4.12 shows the impact to the time needed to simulate the booting of the Operating System. We have grouped the simulator executions by the simulated number of processor cores, normalized the execution time to the simulator without dynamic testing, and then calculated the geometric mean. The results indicate that dynamic testing reduces the simulator speed by 20% on average, with a very small standard deviation. Since there are no transactions during the booting of the OS, there is almost no penalty for doing the empty calls to the HTM testing code.

Figure 4.13 shows the performance impact of dynamic testing during application execution. We have grouped the simulator executions by the simulated application, normalized the execution time to the simulator without dynamic testing,

and then calculated the geometric mean. According to the evaluation, dynamic testing reduces the execution time between 10% and 20%, which is relatively less than during the OS booting. The reason is that the basic simulator is now more complex and simulates an HTM protocol. We can see that, while dynamic HTM testing does introduce some overhead, the total increase in the simulator execution time is generally below 20%.

In both testing examples, dynamic runtime testing would extend a 10 hour simulation to less than 12 hours on average. Taking into account that writing the simulator and the simulator test suite may take many person-months, we consider the performance impact of dynamic testing to be more than acceptable.

4.6 Our Experience With Dynamic Runtime Testing

It is commonly believed that the earlier a defect is found, the cheaper it is to fix it [45]. Our experience is certainly in accordance with this popular belief. We have developed the dynamic testing methodology out of necessity. Making a cycle-accurate architectural simulator is certainly not easy and, as any other software development, it is very prone to errors.

The original cache coherence protocol in M5 simulator is bus-based, which does not scale well beyond 8 cores (or 16 cores as a maximum). We have replaced the base M5 cache coherence with MESI directory-coherence protocol, known to scale well even with more than 64 processor cores. Our directory-based coherent caches hold both line addresses and data, which means that a bug in the cache-coherence protocol would cause wrong data to be provided by caches. Thanks to using dynamic testing, we were able to complete the implementation of caches in under 3 months, and to have much stronger confidence in the correctness of our implementation.

Our first two HTM simulators did not implement the dynamic testing methodology. These two simulators were supposed to be used for validating the results presented by LogTM [55] and TCC [30]. After more than 12 man-months spent on simulator development we had to cancel the development, since some simulations were still not terminating correctly, or were giving wrong results. This would jeopardize the objectiveness and the correctness of our measurements. To find

and eliminate bugs, we would have to analyze the execution traces of hundreds of gigabytes, and this task is nearly impossible to be done manually.

Dynamic testing methodology in our following simulators allowed us to significantly reduce the time needed to transition from an idea to getting the evaluation results. The benefits from dynamic testing are two-fold. First, since we knew *exactly* where a bug appeared in the simulator execution, we could quickly detect and eliminate all obvious simulator bugs. This reduced the simulator development time from 12-18 man-months to 3-4 man-months. Second, dynamic testing methodology improved our confidence in the results of our evaluations, since we had a proof that all our HTM simulations were functionally correct.

Three of our HTM simulators have lazy version management and one has eager version management. Although the functionality of these HTMs is different, they all have similar functional equivalents. A fundamental difference between the eager and the lazy HTM is the decision on when to abort a conflicting transaction. In all implementations, a transaction can keep its speculatively modified values private, in a per-transaction buffer, since the speculative values become public after a transaction commits.

4.7 Related Work

Dynamic runtime testing is related to several testing and debugging methodologies of software and hardware. These section describes several related testing and development methodologies.

Conventional debugging methods help discover how and why a bug occurred, but they offer very little help for discovering *whether* and *where* a bug occurred. It is also possible that there is a logical flaw in the simulated protocol. These flaws cannot be detected easily using conventional debugging methods.

To detect bugs, a developer may add assertions to a program [40], to check for illegal values of some variables. However, a developer needs to add assertions manually. This means that assertions detect only the bugs that a developer can anticipate, for example, this value should *never* be zero. However, some bugs produce values that are valid but incorrect. For example, if an assertion checks if a value is non-zero, that assertion would not detect an incorrect value 2 instead of 3.

In result, beside polluting the source code, the assertions detect only a small subset of bugs. Finally, even if an assertion fails (after detecting an illegal value), the bug that caused the illegal value could be millions of cycles before the assertion fails. Discovering *where* in the execution a bug appeared is a difficult problem. Being "efficient in debugging" is directly related to the previous experience in debugging and programming, causing debugging to be closer to an art than to a science. A bug may cause an execution to: (1) fail, (2) terminate with an incorrect result, or (3) terminate with correct result. We cannot underestimate the final case, where a program terminates with correct result even though it has bugs. These bugs values might cause an execution to be shorter or longer than it should be, for example, by causing a wrong number of loop iterations.

In contrast with assertions, which often check values against constant illegal boundary values, dynamic runtime testing provides precise reference values to compare an execution with. In that sense, we can see dynamic runtime testing as assertions with dynamic precise conditions, where the conditions for assertion checks are strict and calculated in runtime, based on the history and the current state of the simulator execution.

Back-to-back testing methodology [77] consists of comparing the execution of two independent and functionally equivalent programs. The programs are compared: (1) statically (for example, by reading the source code), (2) with specially designed functional test cases, and (3) with random test cases. However, in back-to-back testing methodology the developers need to dedicate significant time to creating a large collection of test cases. In contrast, dynamic runtime testing is a small, "write and forget" one-time development effort that autonomously performs tests during entire life cycle of the simulator.

"Co-Simulation" (co-operative simulation) [8] was proposed as a way to accelerate the simulations in Hardware Description Language (HDL). Co-simulation consists of partitioning the simulator into modules and simulating some modules in hardware simulators (HDL) and the rest in software (e.g., C code). The hardware and software modules exchange information in a collaborative manner, using a well defined interface. Since modules simulated in software are much faster than the modules written in a low-level HDL, the simulation can be completed much faster. Co-simulation is sometimes extended for verification, but the problem of

interfacing modules in a heterogeneous simulation platform presents a major issue both in performance and programmability. In contrast, dynamic runtime testing was developed with an objective to provide functional verification and has all simulator components written in the same language, on a homogeneous simulation platform. Having a homogeneous simulation platform allows easier development and testing, stronger integration of simulator modules, and faster execution.

4.8 Conclusions

Dynamic runtime testing can be used for improving stability and reliability of cycle-accurate architectural simulators. With dynamic runtime testing, we verify functional correctness of the simulator automatically, with every simulator execution.

This allows us to change the simulator rapidly, and still be able to find bugs quickly and be confident that the simulator executes correctly. The simulator reports us any potential functional bug, together with the exact time and the circumstances that lead to the bug. The testing method imposes only a minor reduction in simulator performance and, in our case, we have managed to reduce the total time for development and evaluation time for a single simulator from 12-18 person-months to 3-4 person-months.

5

EazyHTM

EazyHTM separates the tasks of *conflict detection* and *conflict resolution*. It performs conflict detection concurrently with a transaction's execution, but defers conflict resolution until either: (1) a transaction tries to commit, or (2) until a conflicting transaction commits (at which point the tentative conflict becomes unavoidable). Unlike traditional eager conflict detection and resolution this means there is no need to anticipate which transaction is more likely to commit. Unlike traditional lazy conflict detection this means we can avoid commit-time validation, and in turn can have simpler and faster commits.

In the following section we introduce the basic EazyHTM protocol. After that, we explain further optimizations and extensions in Section 5.2. We overview the micro-architectural modifications in Section 5.3, and discuss the experimental results in Section 5.4.

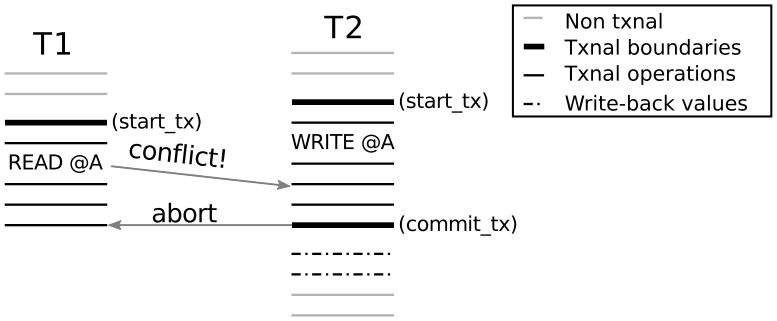


Figure 5.1: EazyHTM conflict detection and resolution: conflicts are detected eagerly, but transactions continue "racing" until one of them commits. The first to commit aborts the racing transaction.

5.1 EazyHTM: Basic Protocol

The EazyHTM protocol operates by cores sharing information within each other on every possible conflict, but not immediately aborting or stalling a transaction.

In Figure 5.1, a transaction T1 reads a cache line that has been speculatively modified by transaction T2. The transactions detect this situation, and note the conflict until they terminate their execution. Conflicts are always in one direction, i.e. the transaction that modifies a cache line has a conflict with, and can abort, the one that reads the same line.

Since all conflicts are detected while a transaction is running, once a transaction (say T2) is ready to commit, it knows exactly which transactions need to be aborted to maintain the system consistency. Therefore, an abort message is sent to all the conflicting transactions and once all conflicting transactions confirm their abort, speculatively written values are published. Both the abort request and the acknowledge are sent over the core-to-core interconnect to the corresponding core. The messages do not have to pass through a centralized router, instead they hop from one core to another until they get to the destination. Conflict resolution only requires the participation of processors involved in the conflict and does not involve the directory.

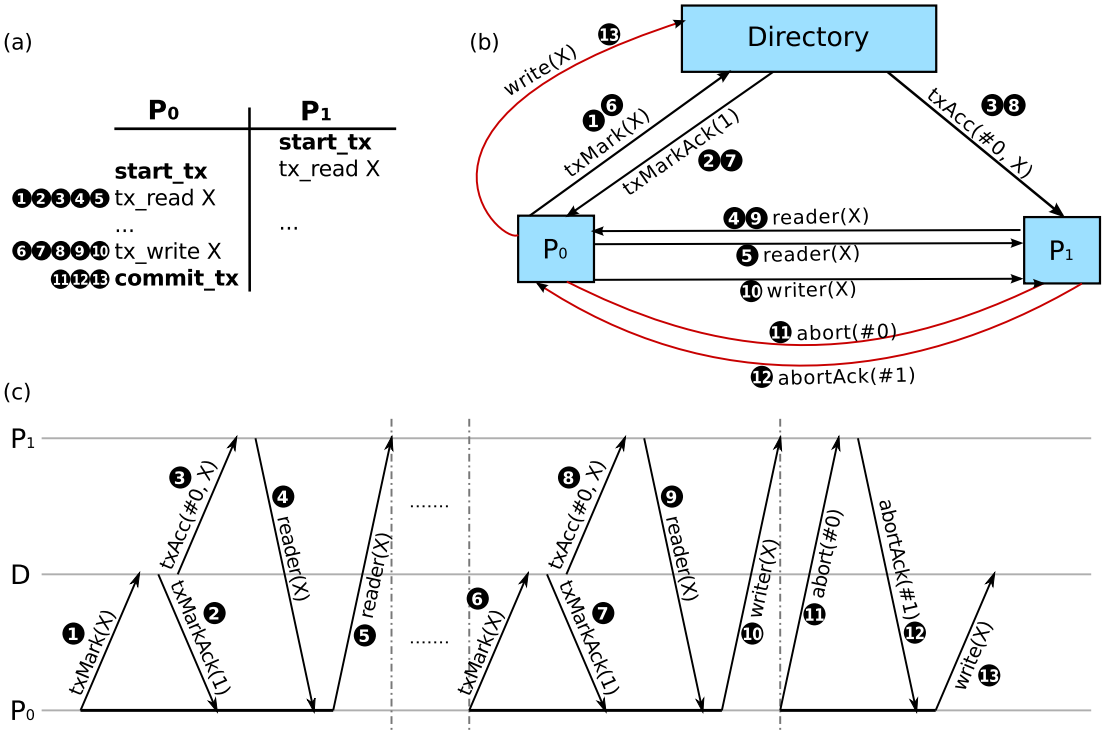


Figure 5.2: Messages for conflict detection and resolution – three memory accesses, one commit and one abort message: **(a)** Concurrent conflicting transactional executions on processors P0 and P1; **(b)** Messages exchanged between two conflicting transactions; messages are numbered in the order they are sent; **(c)** Time diagram of the message exchange, with time going from left to right. Thick horizontal line segments on P0 line mark a single executed instruction.

5.1.1 Conflict Detection

EazyHTM bases its conflict detection on the existing cache coherency functionality, currently used for non-transactional code to ensure that no races occur on shared accesses. Concretely, in a directory based implementation of cache coherence, this extension of the functionality only requires that the directory responds to one new message corresponding to a transactional access. Like any other directory protocol, this protocol is completely transparent to running code.

When a transaction accesses a line, it sends the directory a special request $txMark(addr)$, regardless of the access being a read or a write. The directory handles this request almost like a read request in an ordinary directory protocol. It marks the read for the line, and after allocating the line in shared mode, responds with an acknowledgement indicating how many other sharers the line has.

We illustrate message flow and conflict detection through an example of two transactions running on the processors P0 and P1 (Figure 5.2). In this example, the transaction in P0 starts, and performs a read which does not conflict with the read in P1. It then does a write which conflicts with the transaction in P1. Finally the transaction in P0 commits while aborting the transaction running in P1.

When P0 speculatively reads line X, a $txMark(X)$ message is sent to the directory ❶. As with typical MESI protocols, a processor only sends $txMark(addr)$ on the first access to the line in the current mode (read or write). For subsequent transactional access to the same line, the processor uses values in its private cache. If the core has previously sent a $txMark$ due to a transactional read and now requires to write a value, it resends the message to detect potential new races.

The directory first acknowledges P0 with a $txMarkAck(1)$ message, where the parameter "1" indicates the current number of accessors for that cache line ❷and then sends a $txAccess(\#0, X)$ message ❸to all the other accessors, in this case P1. This is possible since the directory keeps track of all speculative (transactional) or non-speculative accesses to all cache lines. As P1 previously accessed the cache line X, the directory knows the list and the count of all cache line sharers. In the following text we are going to use term "accessor" instead of a "sharer", to represent both non-transactional and transactional cache line sharers, which might have read and/or modified the line.

On receiving $txMarkAck(1)$, P0 waits until it receives the specified number of messages from all other accessors (in this case, it waits until one message is received).

Meanwhile, $txAccess(\#0, X)$ initiates a point-to-point communication between the old accessor, P1, and the new one, P0. Note that P1 knows that the new accessor is P0 because of the first parameter in the $txAccess$ message. The list and explanation of all the messages that can be interchanged between processor cores is given in Section 5.1.5.

Continuing with the example of Figure 5.2, P1 informs P0 that it is a speculative reader of the line X by sending the $SR(X)$ message ❹. When P0 receives a message, it sends a response message $SR(X)$ to P1❺. Now both transactions know the exact access mode of both transactions for the line, and both of them know whether there is a conflict between them or not. In this specific case, since both accesses

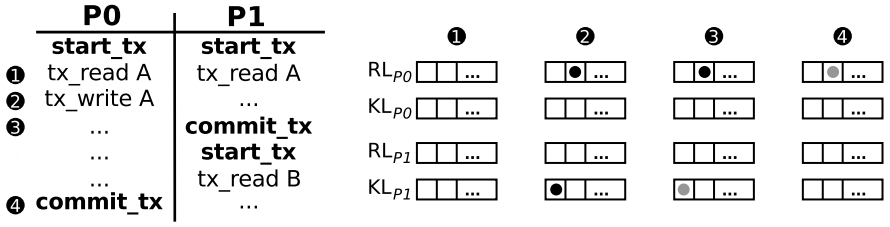


Figure 5.3: Racers-list (RL) and killers-list (KL). The racers-list records transactions that need to be aborted when this transaction commits. The killers-list records transactions that have the permission to abort this transaction.

are reads, there is no conflict between transactions.

In the example, a conflict occurs when P0 speculatively writes to the line X. P0 sends a $txMark$ message ❻ to the directory, which causes the directory to send exactly the same messages as in the non-conflicting situation, ❼ and ❽. At this moment, a point-to-point communication starts again, with P1 sending a $SR(X)$ message to P0 ❾. P0 responds with its access mode to the line, and sends a $SW(X)$ message to P1 ❿

5.1.2 Tracking Possible Conflicts

The *racers-list* on processor Pi maintains a list of other processors that run transactions which conflict with Pi's current transaction. This over-approximates the set of transactions that need to be aborted when Pi's transaction commits (e.g. a conflicting transaction on another processor Pj may have aborted, and a different non-conflicting transaction started in its place).

To avoid false-aborts in this kind of case, each processor maintains a *killers-list* of processors that are allowed to abort its transaction.

Both the racers-list and the killers-list must be cleared at the start of each transaction.

Figure 5.3 presents an example with races. Initially, both racers-list and killers-lists are empty on both processors. After event ❶, the lists are still empty since both accesses were reads. When P0 executes the write instruction ❷, it receives a $SR(X)$ message from P1. This adds P1 to P0's racers-list. Also, since P1 gets a $SR(X)$ message from P0, it adds P0 to its killers-list ❷.

After P1 commits the current transaction ❸ and starts a new transaction, the

killers-list is cleared. This prevents P0 from aborting the next transaction running on P1 ❹, unless a new race is established.

5.1.3 Committing a Transaction

Unlike HTMs with lazy conflict management HTMs [23, 32], always has knowledge which transactions are valid, and how to preserve the validity of transactions. At commit time, the racers-list and killers-list provide the information for a transaction to know which other transactions it is conflicting with it; further commit-time validation is not required. To commit a transaction in EazyHTM, we only need to ensure the termination of transactions from its racers-list. The transactions from the racers-list can either abort or commit.

Commit with conflicts and aborts: When the transaction running on P0 in Figure 5.2 reaches the commit instruction, it has to abort all the conflicting transactions in order to ensure isolation. When P0 is ready to commit, it first sends an $abort$ message to all processors from its racers-list (P1 in this case) ⓫ P1 aborts only if P0 is in its killers-list. However, in both cases P1 sends an acknowledge ⓬ to P0's abort request.

Once P0 has received $abortAck$s from all conflicting cores, it enters the committing state where it is guaranteed to commit successfully. During this period the transaction cannot be aborted and responds to all possible killers with an $abortNack$.

The processor writes all speculatively modified cache lines serially to the shared memory in the usual manner: acquires exclusive access from the directory, for each line in its write-set ⓭ which in turn invalidates copies held by all other accessors. After publishing all the cache lines, it exits the committing procedure and continues normal execution. Also, see Section 5.2 for optimizations for this process.

Commits without conflicts: In case no conflicts are present, EazyHTM allows all non-conflicting transactions to commit in parallel. Figure 5.4 shows the execution of two non-conflicting transactions running on P0 and P1. Since the transactions datasets are disjoint, the directory does not send any $txAccess$ message to processor cores during executions. Therefore, there are no core-to-core messages between P0 and P1. Both transactions have empty racers-lists at the moment of

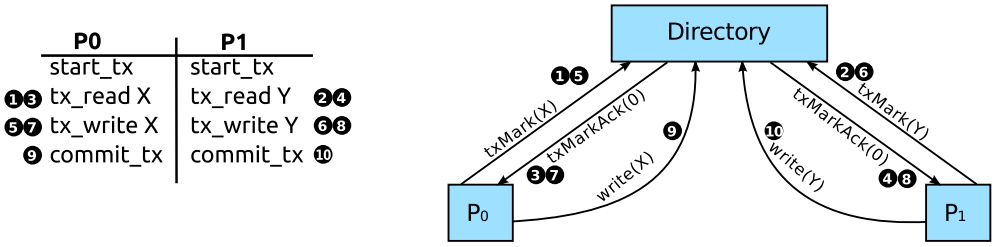


Figure 5.4: Committing without conflicts: transactions accessing different cache lines do not incur any extra communication between them.

commit (not shown in figure).

Figure 5.4 also shows the low overhead EazyHTM imposes on non-conflicting accesses. In particular, a non-conflicting read/write results in the same number of messages as a normal, non-transactional read/write.

Racing commits: Though uncommon in practice, it is possible for multiple racing transactions with mutual races to reach the commit instruction at *exactly* the same time. In this case, transactions would receive an abort request from a transaction that they just sent an abort request to (and did not receive an acknowledgement from). One of the transactions must now abort to allow the other to proceed. EazyHTM breaks ties in this case by allowing the transaction running on a lower cpuid to win and commit. This transaction sends an $abortNack$ to the transaction running on the core with the higher cpuid, which responds with an $abortAck$ and aborts itself. This situation is extremely rare, so we use a simple criterion. Note that progress is still guaranteed. Random, round robin and a number of other tie-breaking policies may be easily added to eliminate the possibility of pathological cases leading to starvation.

5.1.4 Aborting a Transaction

With EazyHTM, a transaction may only be aborted for one of the following reasons:

1. In response to an abort request sent by another transaction.

2. On exceptions or interrupts. In general, hardware transactions are small enough to complete between occurrences of exceptions or interrupts. Even

TLB misses, which are unavoidable, become less important as the TLB warms up, and do not significantly affect our system.

3. When non-transactional code modifies a cache line being accessed in a transaction: this allows us to support strong atomicity, as defined by [14]. The feature is provided by detecting a cache coherency invalidation message from the directory and aborting if a part of the transaction gets invalidated by the directory.

Aborting a transaction in EazyHTM discards all the speculatively performed updates and restarts the transaction execution. Since we implement lazy version management, caches can quickly invalidate all speculative changes. The racers-list and killers-list (see Section 5.1.2) are also cleared on abort.

Once all speculative changes are discarded and the lists cleared, the register file is restored to its previous state, saved just before the beginning of the transaction, and the control flow is reset to the first instruction.

5.1.5 State-Message Table of the EazyHTM protocol

For completeness, we present a complete transactional state table of a processor core (Table 5.1 on Page 69). Each cell in the table describes the actions performed by a core upon receiving a message, depending on its current transactional state. The rows represent current state. The columns represent the incoming message from another processor core or from a directory. Dashes indicate impossible combinations. We define each CPU core state as follows:

- *Active*: A transaction is being executed on the core.
- *Ready to commit*: The transaction has executed all the code within the atomic block and is in the process of aborting all racing transactions. Lasts between the beginning of the $commit$ instruction and the reception of the last $abortAck$ or $abortNack$ (if any).
- *Committing*: The processor core has aborted all racing transactions and is now committing speculative changes. Once entered in this state, the transaction is invincible: it cannot be aborted.

State \Message		SR(@)⋆	SW(@)⋆	SRW(@)⋆	nonTXnal(@)	tryLater(@)	txAccess(@)	abortAck(#)	abortNack(#)	abort(#)
Active TX	NoTx	-	-	-	-	-	nonTXnal(@)	-	-	(1)
	SR	SR(@)⋆	SR(@)⋆; KLR	SR⋆(@); KLR	nop	REDO INSTR	SR(@)	-	-	
	SW	SW⋆(@); RCR	SW⋆(@)	SW⋆(@); RCR	nop	REDO INSTR	SW(@)	-	-	
	SRW	SRW⋆(@); RCR	SRW⋆(@); KLR	SRW⋆(@); RCR; KLR	nop	REDO INSTR	SW(@)	-	-	
Ready to commit		-	-	-	-	-	tryLater(@)	(2)	(3)	(4)
Committing	NoTx	-	-	-	-	-	NonTXnal(@)	-	-	(5)
	SR	-	-	-	-	-	NonTXnal(@)	-	-	
	SW	-	-	-	-	commit, nonTXnal(@)	-	-		
	SRW	-	-	-	-	commit, nonTXnal(@)	-	-		
Aborting		-	-	-	-	-	nonTXnal(@)	-	-	abortAck(#)
Inactive TX		-	-	-	-	-	nonTXnal(@)	-	-	abortAck(#)

Table 5.1: State-message table of a core. The current state is on the left, and incoming messages are on the top. Each cell shows the action to be performed. Only message *txAccess* comes from directory, all others come from other cores in the system. Legend:
@ cache line address and processor core id
# processor core id
- error state
⋆ the response messages are marked as such, so that they do not get responded again by a receiving processor.
KL killers list
RCR set a bit in racers list
KLR set a bit in killers list
CRL clear racers list
(1) abortAck(#); if sender-core-id in KL: abort
(2) CRL; if RL == 0: enter committing
(3) wait all pending abortAck or abortNack and then abort
(4) if my-core-id > sender-core-id: {abortAck(#), abort} else: {CRL, abortNack(#)}
(5) if sender-core-id in KL: {abortNack(#)} else: {abortAck(#)}

- *Aborting*: The transaction has received an $abort$ message and is processing it, i.e. flushing speculative changes.
- *Inactive Tx*: The core is not executing transactional code.

Some messages are related to one cache line and different actions may be taken depending on whether this line is present in the read set, write set or neither. Therefore, the states *Active* and *Committing* have sub-states:

- *NoTx(addr)*: Neither *SR* nor *SW* bit are set for the line address *addr*,
- *SR(addr)*: The *SR* bit is set for the address *addr*, that is, the address is in the read set of the transaction,
- *SW(addr)*: The *SW* bit is set for the address *addr*, that is, the address is in the write set of the transaction,
- *SRW(addr)*: Both *SR* and *SW* bits are set, that is, the address is in both read and write set of the transaction.

The sub-states *SR* and *SW* are set in the private cache by the processor core before a transaction sends the request for the line to the directory. This makes sure that any incoming message regarding that line is handled properly.

In the following text we list and explain all messages that our approach introduces.

- $SR(addr, cpuid)$: The address "addr" is in the read set of a processor core cpuid.
- $SW(addr, cpuid)$: The address "addr" is in the write set of a processor core cpuid.
- $SRW(addr, cpuid)$: The address "addr" is in the read and write set of a processor core cpuid.
- $nonTXnal(addr, cpuid)$: The address "addr" is not accessed transactionally by a processor core cpuid.

- $tryLater(addr, cpuid)$: P1 is trying to access address transactionally but P0 cannot respond at the moment.
- $txAccess(addr, cpuid)$: This message comes from the directory rather than from another core. It indicates the receiver that the core cpuid is accessing address "addr" transactionally, and that they should communicate and exchange their access mode.
- $abort(cpuid)$: Request to abort, sent from the processor core cpuid.
- $abortAck(cpuid)$: Abort acknowledgement (ACK), sent from the processor core cpuid.
- $abortNack(cpuid)$: Abort negative acknowledgement (NACK), sent from the processor core cpuid.

Certain state-message combinations are worth explaining since they may not be intuitive. For example, when a transaction is in the *Committing* state and gets a $txAccess(addr, cpuid)$ message, it replies with a $nonTXnal(addr, cpuid)$ message (previously committing the line if it has been speculatively modified). This behavior is so defined by the "critical cache line first" optimization, explained in Section 5.2.

5.1.6 Proofs of protocol correctness

Let T_i and T_j be two transactions in the system. Let R_i and R_j be the racers-list of T_i and T_j respectively. Let K_i and K_j be the killers list of T_i and T_j respectively.

Lemma 5.1.1. *A transaction's racers-list R_i does not miss any conflicts from T_i to other transactions.*

Proof. R_i is cleared only when T_i aborts, so no conflicts can be lost during the lifetime of T_i. We now need to show that all conflicts from T_i to T_j are marked in R_i. From Table 5.1, T_j sends messages SR and SW to T_i on all speculative reads and writes. This means that T_i will be notified of all speculative accesses made by T_j. We need to show that R_i will represent all cases where T_i writes to a line, while T_j reads or writes.

Incoming *SR* message from T_j: (1) T_i only read the line: there is no conflict from T_i to T_j and R_i is not modified, (2) T_i is a speculative writer: conflict from T_i to T_j exists, and T_j is added to the R_i.

Incoming *SW* message from T_j: (1) T_i only read the line: there is a conflict from T_j to T_i but not in the opposite direction, from T_i to T_j. That is, T_i does not have to abort T_j if T_i is to commit, so R_i is not modified, (2) T_i is a speculative writer: there is conflict from T_i to T_j, and it is marked in R_i.

We have shown by exhaustion that R_i marks all write-read and write-write conflicts between T_i and T_j. Therefore, there is no case where T_i conflicts with T_j, without having the case marked in R_i. □

Lemma 5.1.2. *A transaction's killers-list K_j marks real conflicts.*

Proof. We need to show that only real conflicts are marked in K_j. Transaction T_j starts with an empty K_j.

Incoming *SR* message from T_i: there is no real conflict from T_i to T_j, and K_j is not modified.

Incoming *SW* message from T_i and T_j speculatively read or modified the line: there is a conflict from T_i to T_j, and T_i is added to K_j.

We see that T_j adds T_i to K_j if and only if there is a real conflict from T_i to T_j. □

Lemma 5.1.3. *A combination of R_i and K_j ($R_i \cap K_j$) precisely represents the conflicts between T_i and T_j.*

Proof. Lemma 5.1.1 and 5.1.2 show that R_i represents all conflicts between T_i and T_j made during the lifetime of T_i. On the other hand, K_j represents only the conflicts occurred between T_i and T_j made during the lifetime of T_j.

Therefore, a combination of R_i and K_j ($R_i \cap K_j$) represents only the real conflicts that are created while both T_i and T_j are executing. □

Lemma 5.1.4. *When a transaction T_i is ready-to-commit, it attempts to abort transactions $T \in R_i$ and no other transactions*

Proof. By protocol definition. □

Lemma 5.1.5. *If a transaction $T_i \notin K_j$, T_i may not abort transaction T_j.*

Proof. From Table 5.1, if $T_i \notin K_j$, T_j sends an $AbortAck$ to T_i on receiving an $Abort$ request, but ignores the request, and *does not abort*. □

Theorem 5.1.1. *For all T_j aborted by, or aborting, a transaction T_i, T_j conflicts with T_i.*

Proof. From Lemma 5.1.4 and 5.1.5, $T_j \in (R_i \cap K_i)$. By Lemma 5.1.3, $R_i \cap K_i$ contains only transactions that conflict with T_i. □

Theorem 5.1.2. *Conflict-free transactions may commit in parallel.*

Proof. From Section 5.1.3, committing a transaction T_i involves two distinct stages, which are performed in serial order. In the first stage, the transaction aborts a number of transactions. In the second, all modified values are committed.

The second stage is clearly independent of other transactions, and may be performed in parallel if the architecture allows it. By Theorem 5.1.1, the first stage only affects transactions that conflict with T_i, and non-conflicting transactions may proceed in parallel.

Therefore, non-conflicting transaction can commit in parallel. □

5.2 EazyHTM: Optimizations

In this section we introduce a series of optimizations to the basic EazyHTM protocol. We present several optimizations to the basic EazyHTM protocol, classified into two groups: (1) optimizations to the commit operation, and (2) optimizations of the conflict detection.

The first group, i.e., the optimizations of the commit operation, includes two particular optimizations: (1) write-back publishing of speculative changes, which accelerates the actual commit operation on a given core, and (2) publishing critical cache-lines first, which accelerates the execution of transactions on other cores.

The second group, i.e., the optimizations of the conflict detection, reduce the unnecessary conflict-detection traffic in the system. Here, we eliminate the conflict detection for transactional lines that are accessed in a certain way. We present three optimizations: (1) core-local filtering of exclusive lines, (2) directory-level filtering of read-only lines, and (3) core-local filtering of read-only lines.

5.2.1 Commit: Write-Back Publishing of Speculative Changes

Following other lazy version management HTM proposals [19, 20, 51, 52], EazyHTM also implements the write-back commit optimization. Lazy version management HTMs have to publish (in some way) their speculatively modified lines when transaction commits. What can be done as an optimization is to publish only the addresses, and to leave the updated cache line contents in private caches. This is called write-back commit.

EazyHTM implements write-back commit in the following way. During the transaction execution lines are augmented with speculative read/write status bits. When the transaction comes to a commit, it aborts all racing transactions, receives confirmation of their aborts and then asynchronously (without waiting for confirmation of every message before sending the next one) sends the addresses of all speculatively modified lines to the directory. The directory marks all these lines as exclusive to the processor core. At the same time, in private caches, the line is marked as non-speculatively modified.

A write-back of the cache line contents is performed when either: (1) another core requests the line later in time, or (2) a line has to be modified speculatively again, by another transaction in the same processor core. In this case, the cache line access mode is reduced from exclusive or modified to shared, and the execution continues.

5.2.2 Commit: Publishing Critical-Cache-Lines First

Regardless of how little validation is performed at commit-time, the duration of the commit-phase is bounded below by the time it takes to publish the speculative modifications of a transaction.

However, EazyHTM escapes this lower-bound by using a critical-cache-line-first transaction commit policy. After all transactions that were racing with the current transaction have acknowledged an abort, we take this moment of time (the moment of receiving last acknowledgement) as the unique point in time when the transaction commits.

Once the validation is complete, the transaction starts publishing all its speculatively modified lines, in some arbitrary order. If, during this phase, any other

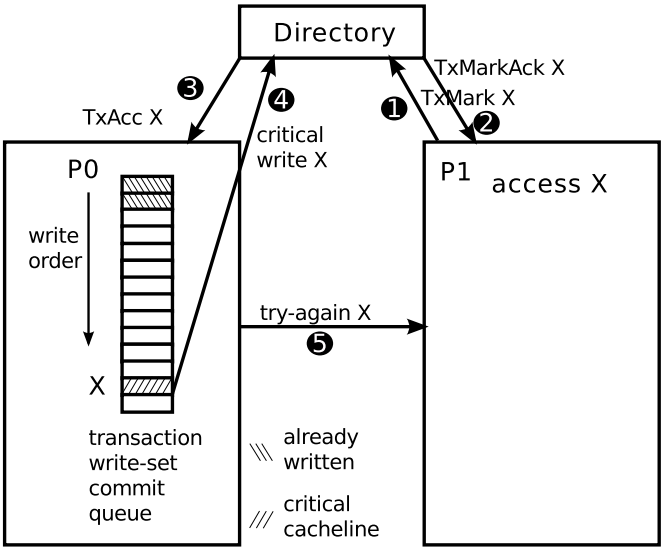


Figure 5.5: Critical cache line First illustration; while P0 is committing values, P1 requests a not-yet-written line X; this causes X to be written first, out of normal commit order

transaction wishes to access some not-yet-committed cache line from the committing transaction, the committing transaction will get notified from the directory. The commit order now gets changed, and the critical cache line is committed first.

After this, a $tryLater(addr)$ response is sent to the requester. When the requester receives this message, it requests the cache line again. This time it gets the new value from the shared memory and a $nonTXnal(addr)$ response from the committing transaction since the line will not be in the write set anymore. We have effectively saved the stalling time of the requester which would be spent in waiting on the committing transaction to finish. An illustration of this situation is shown on Figure 5.5. Since T2 receives a $nonTXnal(addr)$ message (not $tryLater$ or $abort$), it appears that T1 has finished committing, and so commits seem to be instantaneous.

5.2.3 Conflict Detection: Core-Local Filtering of Exclusive Lines

An exclusive line cannot create any conflicts between transactions, and informing the directory of transactional accesses to exclusive lines is completely unnecessary. When a transaction accesses an exclusive line, and sends a message to the directory,

the directory will always respond with the $txMarkAck(0)$ message, indicating that there are no other sharers of the line.

One of the line states with the standard MESI protocol is "Exclusive". If Eazy-HTM is implemented on top of MESI (or compatible) protocol, it can leverage this information and avoid sending unnecessary messages to the directory.

5.2.4 Conflict Detection: Directory-Level Filtering of Read-Only Lines

If a transaction reads a line and all the other accessors are readers, then messages exchanged between them will be informing one another about their reader-reader status. No modifications will be done neither to the racers list nor to the killers list. Therefore, these messages can safely be avoided.

In order to eliminate these messages, we also propose a directory-level filtering of these messages. We add an extra bit per directory entry. This "Transactionally Dirty" (TD) bit represents whether the cache line is in the write-set of at least one active transaction or not. To distinguish between transactional reads and writes, the $txMark(P,X)$ message to the directory has to be split in two messages: $txMarkRead(P,X)$ and $txMarkWrite(P,X)$. The directory handles these messages as follows.

- $txMarkRead(P,X)$: (1) if the TD bit is set, a message $txAccess(P,X)$ is sent to every accessor, as explained in Section 5.1. (2) If the TD is zero, no messages are sent to the other accessors, because they are all readers.

- $txMarkWrite(P,X)$: the directory sets the TD bit, and a $txAccess(P,X)$ is sent to every accessor (if any).

The TD bit is cleared on a non-speculative write to the cache line; i.e. when either (1) a transaction commits and thus writes all speculative values to the shared memory, or (2) when a regular, non-transactional code writes to the line. Note that although this modifies the directory structure, the protocol is not changed in an extensive way.

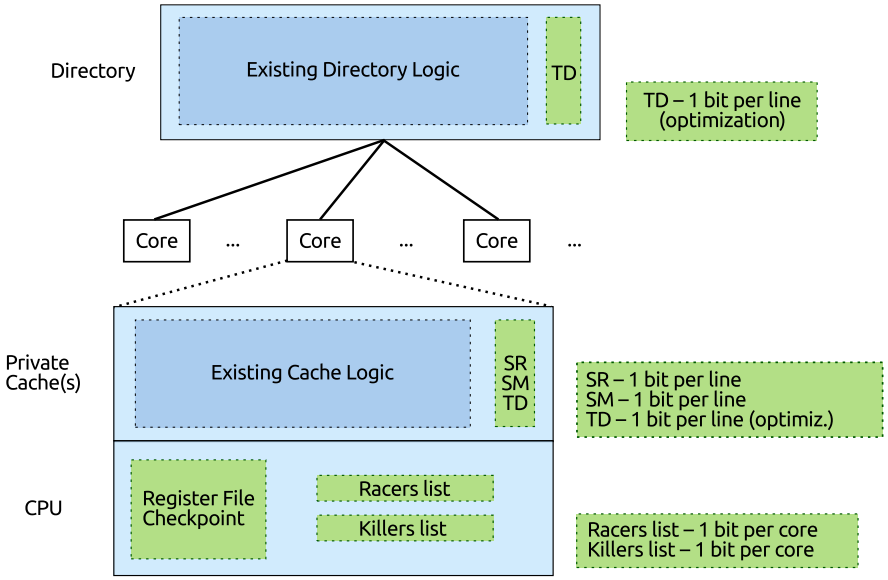


Figure 5.6: An overview of the EazyHTM hardware modifications

5.2.5 Conflict Detection: Core-Local Filtering of Read-Only Lines

The previously described optimization of filtering out the conflict detection for read-only lines can also be applied to processor cores. The goal is to, not only eliminate the core-to-core messages, but also the core-to-directory messages for read-only lines. In this case, the TD bit is added in L1 caches, and *not* on the directory level.

We can do this in the following way. If a core speculatively writes to a shared line, all sharers mark the line as "TD". All future speculative accesses to such line will invoke a regular EazyHTM conflict detection. For the lines that are not marked as TD, the accesses do not invoke any conflict detection. The TD bit is cleared when there is a non-speculative write to the line.

5.3 Micro-architectural changes

This section introduces the hardware changes which are required for EazyHTM. Described hardware changes support both the basic EazyHTM protocol and the optimizations described in the Section 5.2. A graphical representation of those changes is shown in Figure 5.6 where:

Register file checkpoint: keeps a snapshot of the register file. The snapshot is taken at the beginning of the transaction. It is used to restart the transaction's execution in case it aborts.

Racers-list: stores a list of all transactions that have to terminate execution before this one can commit. It is implemented as a simple bit vector (bitmap), with one bit per core. Detailed explanation of Racers-list is given in Section 5.1.2.

Killers-list: stores a list of all transactions that are allowed to abort the transaction executing on this core. It is also implemented as a bitmap, with one bit per core. Detailed explanation of its functionality is given in Section 5.1.2.

Cache support: EazyHTM protocol requires tracking transactional accesses to lines from private cache. Thus we extend the private caches with two extra bits: a speculatively-read (SR) bit indicates that the associated cache line has been read by the currently running transaction, and the speculatively-written (SW) bit indicates that the cache line has been modified by the current transaction. Multiple levels of private caches are possible, provided that they all track this information. An optional TD bit can be added for the optimization core-local filtering of read-only lines, described in Section 5.2.5.

Directory Support: As commented in Section 5.2.4 we add a transactionally dirty (TD) bit per directory entry. This bit marks if a cache line has been speculatively modified by any transaction since its last non-speculative modification. Note that this modification is independent of the number of processors (i.e. if the system had more cores, the TD would still be one bit per line).

5.4 Evaluation

In this section we evaluate the performance of EazyHTM using the STAMP benchmark suite [20]. We describe our simulation environment in Section 5.4.1. Then, in Section 5.4.2, we evaluate EazyHTM and each of its optimizations, and compare them with a perfect-lazy HTM. The perfect-lazy HTM does not have any overhead or latency for conflict detection. We also compare with a variant of perfect-lazy HTM that has an instant commit with zero latency.

Processor(s)	1-32 sequential in-order cores at 2 GHz
L1 data cache	writeback, private, MESI, 32 KB, 4-way, 64B line, 2 cycles hit
L2 cache	writeback, private, MESI, 512 KB, 8-way, 64B line, 8 cycles hit
L3 cache	writeback, shared, MESI, 16 MB, 8-way, 64B line, 16 cycles hit
Main memory	MESI based directory, 200 cycle latency
ICN	2D Mesh, 3 cycles per hop

Table 5.2: Baseline EazyHTM Simulator Configuration

5.4.1 Simulation environment

To evaluate the performance of EazyHTM, we compare it with lazy (instead of eager) conflict management HTM, since there is a general agreement that lazy HTMs have better performance than eager conflict management systems [16, 68]. We show the baseline configuration of the EazyHTM simulator used in our evaluation in Table 5.2.

To minimize the occurrence of overflowed transactions and the performance penalties due to limited hardware resources, the processor cores in our simulator have large private caches. In our configuration each processor has inclusive private L1 and L2 caches. This resulted in few overflow-related transaction aborts. This observation matches the one presented by other researchers, for example [23].

All instructions in our simulator have 1 cycle latency except those that access memory, where the latency of an instruction is increased by the value returned by the memory subsystem (caches, interconnection network, and the main memory). The directory protocol we implemented for EazyHTM evaluation is MESI-based. The directory in EazyHTM is logically placed one level higher than the private L1 and L2 caches. We add one more level of caches, L3, that is shared between directory cores, and memory is equally accessible from all processors and all memory addresses have the same access latency.

The topology of our core-to-core ICN is 2D mesh. This topology has technologically low cost, complexity and power consumption while it provides modest performance [42]. More advanced interconnection topologies, such as 2D torus or 3D torus/mesh, would likely be faster and result in better EazyHTM performance, by reducing the average latency and hop count between cores. The number of hops between any two cores on the die is determined by the ICN, and the assumed

latency per hop is 3 cycles.

5.4.2 EazyHTM Evaluation Results

We evaluated EazyHTM proposal using nine different STAMP benchmark configurations: Labyrinth, Vacation-Low, Yada, Intruder, SSCA2, KMeans-Hi, KMeans-Low, Vacation-Hi, and Genome, with the parameters shown in Table 3.3 on Page 33. "Hi" and "Low" workloads provide different conflict rates. Since we are only interested in the time spent in the parallel section, all the results pertain to this section only. The time spent in transactional execution differs significantly from benchmark to benchmark. We show it in the Table 5.3, for all evaluated applications.

Bayes was not included in our study since it has non-deterministic behavior (i.e., its execution time does not necessarily depend on the speed of execution), and it also has extremely large transaction that does not fit into private caches. Labyrinth has an almost flat speedup curve, since every committed transaction in Labyrinth aborts nearly all other transactions. Consequently, Labyrinth has an almost-serialized execution of transactions. To improve the performance of Labyrinth, STAMP authors proposed the use of early-release. The obtainable speedup is 3-4 times on 16-32 cores, but our current implementation of EazyHTM does not include the support for early-release. For completeness, we still included Labyrinth in the evaluation results.

As we can observe in Figure 5.7, Genome and SSCA2 spend significant time without executing instructions (depicted as *sleeping*). During this time, the processors are put in the quiescent mode until an interrupt occurs. While this behavior is in occasionally present in all benchmark configurations, the time becomes significant only in Genome and SSCA2, due to extensive use of barriers as a way to synchronize the execution of threads. As we can see from the breakdown, quiescent time becomes dominant with larger number of cores (70% of the time for the 32-core execution), and thus completely limits the scalability of applications. The complete time spent in barrier synchronization (not shown in the figures), including the execution of the pthread library functions and associated system calls, takes more than 90% of the execution time for the same 32-core executions.

On the same figure, beside sleeping, the execution time is split into useful and

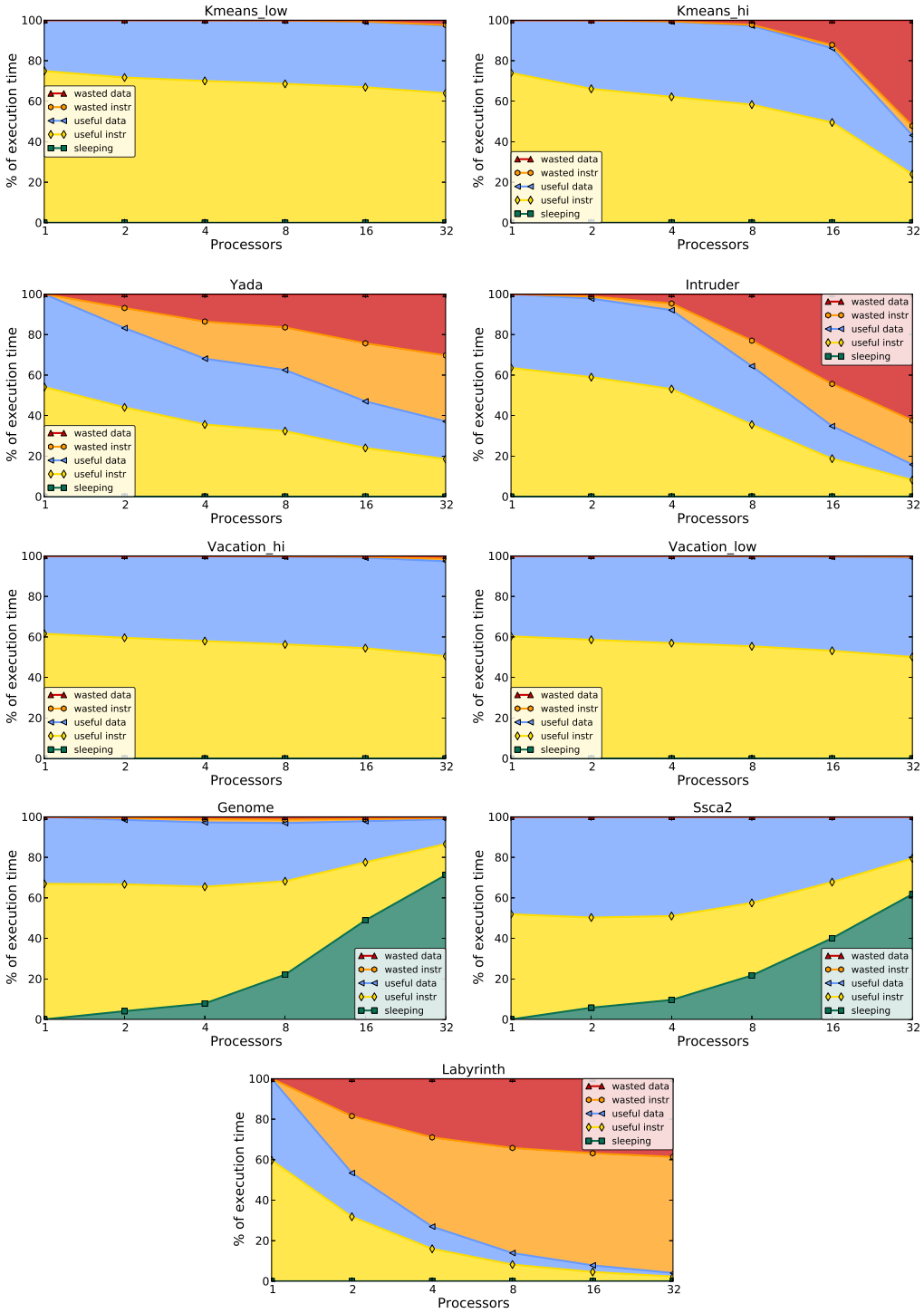


Figure 5.7: The breakdown of the EazyHTM execution time. We consider the EazyHTM configuration with all optimizations activated.

wasted. If transaction commits, its execution time is assigned to *useful*, and if it aborts, its execution time becomes *wasted*. The useful and wasted times are further split into instruction execution (*instr*) and data accesses (*data*). From this breakdown, we can see that Kmeans-Low, and to some extent Yada have good data locality and their committed instructions (useful) spend significantly more time executing instructions than accessing data. Other applications, such as Vacation Low and Hi, do not have very good data locality and spend a significant amount of time in data accesses.

Table 5.3 also shows some statistics: the percentage of time spent in transactions (%TX), the percentage of transactions that aborted (%ABO), and the percentage of transactions that activated the critical-cache-line-first commit optimization (%CLF). We can see that some applications have many invocations of critical-cache-line-first. For example, in Intruder, the over 30% of transaction commits invoke this optimization. While the optimization does not have a significant effect on the execution time, the hardware requirement for the optimization is very low, and because of this it makes sense to include the optimization.

Figure 5.8 shows the execution-time analysis for different levels of optimizations in EazyHTM. As a boundary case of EazyHTM performance, we implemented a MESI-based *Lazy-Ideal* HTM without any overheads. The Lazy-Ideal HTM performs both conflict detection and resolution *instantaneously* (in zero clock cycles), without any extra directory or core-to-core messages. To evaluate the only remaining overhead in Lazy-Ideal HTM – publishing speculative modifications, we provide a variant of Ideal-Lazy HTM which, in addition, acquires all speculative lines in the "modified" state with 0-cycle latency, named Lazy-Ideal-CTX. While having this idealized lazy HTM in hardware is practically impossible, it serves as a good upper bound on the best-case lazy HTM performance and directly evaluates *all* overheads present in EazyHTM.

Beside the two variants of Lazy-Ideal HTM, we present four variants of EazyHTM, that we described in more details in Section 5.2:

1. EazyHTM-base is the basic variant of EazyHTM, which sends a conflict-detection message for all transactional accesses. This increases the latency of transactional operations and, consequently, reduces the performance.

		1 core	2 core	4 core	8 core	16 core	32 core
Kmeans-Low	%TX	3.8	3.8	3.8	3.8	3.8	3.8
	%ABO	0.0	0.6	2.3	3.6	5.3	11.9
	%CLF	0.0	0.0	0.0	0.0	0.0	0.8
Kmeans-Hi	%TX	9.5	9.5	9.5	9.5	9.9	13.0
	%ABO	0.0	1.0	3.2	8.9	26.4	72.9
	%CLF	0.0	0.0	0.0	0.1	1.1	17.6
Yada	%TX	100	100	100	100	99.6	100
	%ABO	0.0	5.2	9.2	15.5	23.7	35.2
	%CLF	0.0	0.0	0.1	0.0	0.3	0.7
Intruder	%TX	39.1	40.6	42.5	51.5	69.6	86.8
	%ABO	0.0	5.6	20.1	43.1	70.3	84.7
	%CLF	0.0	0.0	0.2	1.6	12.0	33.9
Vacation-Hi	%TX	86.0	85.4	84.8	83.6	84.3	83.5
	%ABO	0.0	0.0	0.5	0.7	0.9	4.1
	%CLF	0.0	0.0	0.0	0.0	0.0	0.1
Vacation-Low	%TX	85.9	86.0	86.8	86.0	83.7	80.4
	%ABO	0.0	0.0	0.1	0.3	0.7	1.5
	%CLF	0.0	0.0	0.0	0.0	0.0	0.0
Genome	%TX	97.9	92.7	78.9	57.6	27.6	9.2
	%ABO	0.0	0.6	1.6	3.4	7.7	13.5
	%CLF	0.0	0.0	0.0	0.0	0.1	0.5
SSCA2	%TX	20.0	18.7	16.5	12.9	7.8	3.2
	%ABO	0.0	0.0	0.2	0.4	0.6	1.2
	%CLF	0.0	0.0	0.0	0.0	0.0	0.0
Labyrinth	%TX	99.9	99.9	99.7	100	100	96.1
	%ABO	0.0	22.5	46.5	68.3	82.6	90.4
	%CLF	0.0	0.5	0.5	1.0	0.4	6.3

Table 5.3: EazyHTM Execution Statistics. Legend: **%TX** — Percentage of parallel section time spent inside transactions; **%ABO** — Percentage of aborts (abort rate), calculated as aborts/(aborts+commits); **%CLF** — Number of critical cache line first invocations divided by the number of commits;

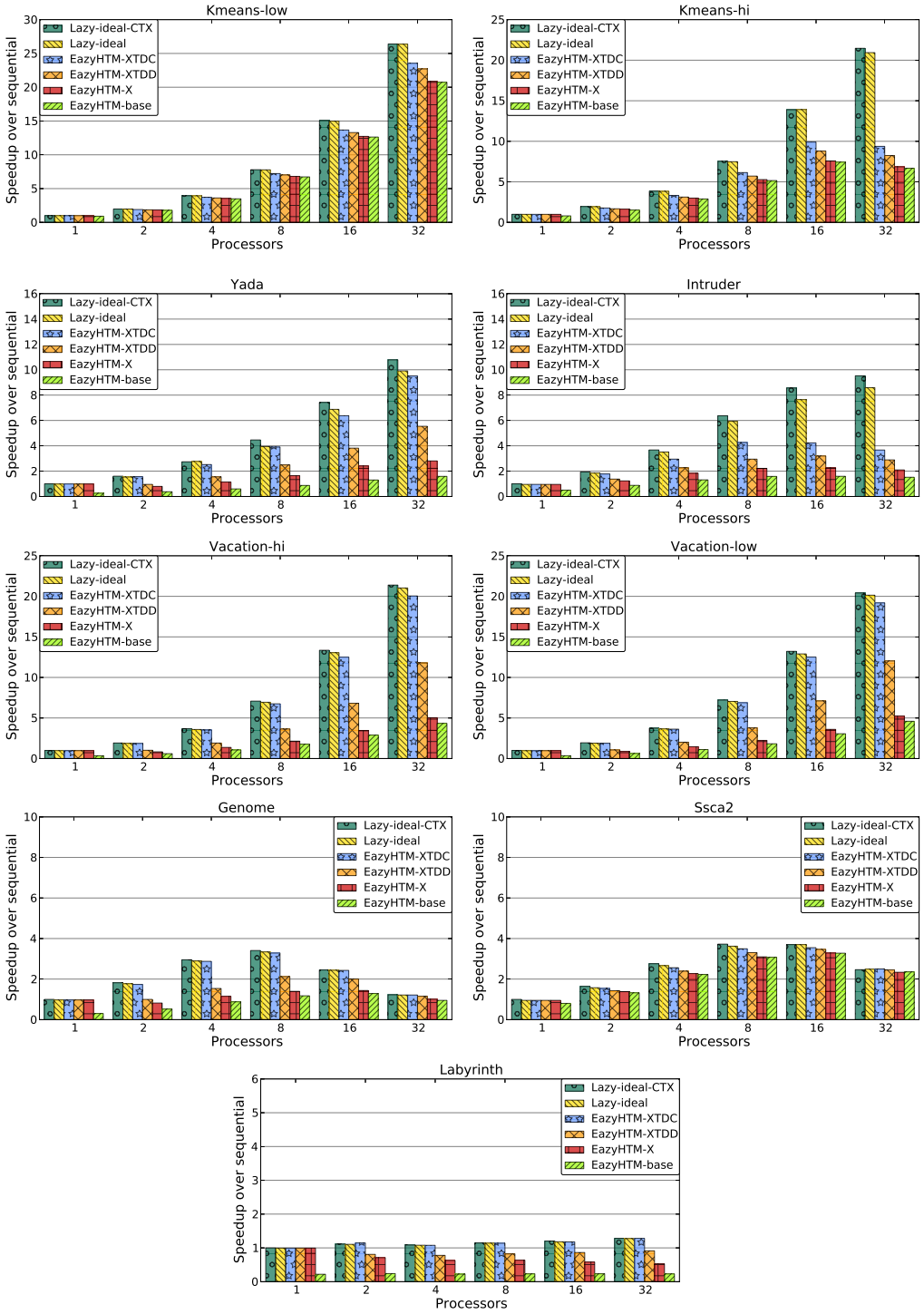


Figure 5.8: The speedup of the STAMP TM benchmark suite applications. EazyHTM-base is the configuration without any optimizations. EazyHTM-X does not do conflict detection for the exclusive lines. EazyHTM-XTDD does a directory-level avoidance of conflict detection for the read-only lines. EazyHTM-XTDC does the same type of filtering only at a core level.

2. EazyHTM-X does not send the conflict-detection messages for the lines that are exclusive to that core.

3. EazyHTM-XTDD also has a filtering bit in the directory, which eliminates the multi-cast conflict-detection messages for exclusive and read-only lines.

4. EazyHTM-XTDC has a similar type of filtering, but provided at the level of a processor core. A conflict-detection message is sent from a core to the directory (and then to other sharers) only if a line has been speculatively modified.

In Figure 5.8, we can see that each optimization improves the performance of at least one application. EazyHTM-X provides the least performance improvement of all optimizations. It helps in Yada, Intruder, Vacation Low, and Labyrinth. This means that very few transactional lines in STAMP benchmarks are exclusive.

Two other optimizations, EazyHTM-XTD and EazyHTM-XRSK reduce the overheads that are due to read-only lines. These optimizations provide much better performance improvements across a wider range of applications. Both optimizations reduce the overheads for the same type of lines, but EazyHTM-XTD does it at the level of directory while the EazyHTM-XRSK does it at the level of a processor core. Some applications especially benefit from the optimization, for example, Yada, Vacation Hi and Low, and Labyrinth. These applications have many transactional lines that are not speculatively modified by any sharer. Avoiding conflict detection for these lines significantly reduces the number of conflict-detection messages, and therefore improves the performance. It is logical and clear from the performance results that EazyHTM-XRSK provides a significantly better performance in cases when EazyHTM-XTD also significantly improves the performance.

EazyHTM shows a performance regression over Lazy-Ideal HTM mainly with one application from the STAMP suite — Intruder. Intruder has very high abort rate. With 32 cores more than 85% of all started transactions get aborted after performing some work (see Table 5.3). This translates to the 85% of entire execution time being wasted, and out of this 44% represents the time spent in the core-to-core communication, 4% spent in sending and receiving directory messages, 39% in cache requests, and 13% in normal execution.

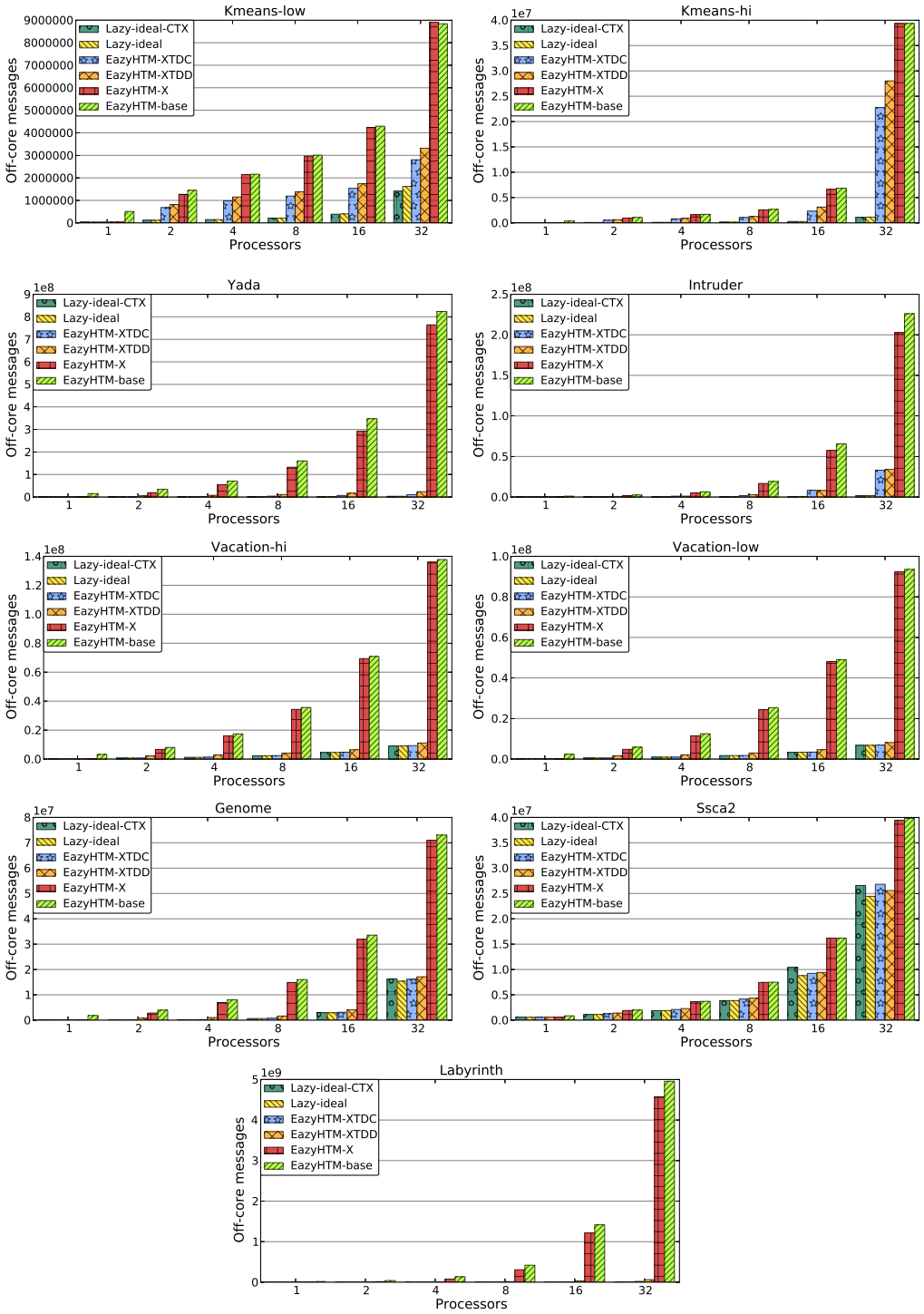


Figure 5.9: Absolute number of off-core messages in EazyHTM variations. Smaller is better. We can see that each optimization significantly reduces the number of off-core messages.

In Figure 5.9, we show the number of messages on the interconnection network, for EazyHTM with various levels of optimizations. We can see that the presented optimizations can significantly reduce the number of additional messages on the interconnection network, introduced by EazyHTM. A particularly good optimization is the filtering of read-only lines. When we use the core-level filtering (EazyHTM-XRSK) of read-only lines, the number of added messages on the interconnection network is insignificant is all configurations except Intruder and Kmeans-Hi with 32 cores.

5.5 Conclusions

EazyHTM explores the advantages of splitting the conflicting management to two distinct actions: conflict detection and conflict resolution. This can be done with very small modifications to the cache coherence protocol on current chip multiprocessors, and allows us to provide a well performing and highly efficient HTM system. EazyHTM makes a good trade-off between hardware complexity, the performance, and the capabilities. To reduce the overheads of the initial EazyHTM design, we applied several optimizations. The optimizations target detecting and eliminating the number of conflict detection messages for the read-only and exclusive cache lines.

Still, even after optimizing the protocol, EazyHTM sends multi-cast conflict detection messages for potentially conflicting cache lines. While multi-cast messages are more efficient than broadcast messages, they are less efficient than point-to-point messages. Our following work is therefore focused on transforming the EazyHTM protocol from using a multi-cast conflict detection to a point-to-point conflict detection protocol.

6

EcoTM: Economical Conflict-Driven Hardware Transactional Memory

EcoTM presents a series of improvements to EazyHTM. We have seen that we significantly reduce the number of conflict detections by adding one bit (TD) to directory entries in EazyHTM. The immediate question we had after that was: how much can we achieve if we add *two bits* to directory entries. After some experimenting with the states that can be represented by these two bits, we got results that changed our opinion about the support for large transactions by HTMs. We have seen that we can efficiently support transactions that are larger than private caches, without inducing false conflicts, common in comparable HTM proposals.

6.1 Introduction

We have seen two possibilities for extending and improving EazyHTM: (1) reduce the network traffic for conflict detection, and (2) supports transactions that over-

flow private caches. An overflowed transaction may impair the performance of an HTM, or complicate the conflict detection in an HTM. Overflowed transactions are either serialized, handled by software, or have their read and write set approximated in Bloom-filter signatures of finite size, determined during hardware design. While the serialization of transactions and software support do not seem as high-performance solutions, Bloom-filter signatures are seen as much better, since they never overflow. However, if more entries are inserted than its fixed size can support, a Bloom-filter becomes saturated and starts returning false-positive hits for conflict detections by other transactions. Because of these false-positive conflicts, large transactions conflict even with unrelated transactions. In result, an HTM with Bloom filters cannot execute large transactions in parallel, which goes against the initial motivation for using an HTM.

The current implementations of Bloom-filter signatures can be improved, but only up to some point. The current work on improving the performance of Bloom-filter is focused on using better hash functions and reducing the number of entries in the signature [65], [80], [48]. However, the current work does not solve the main issue with Bloom-filter-based HTMs: the fixed and finite size of Bloom-filters. To truly support both large and small transactions, we need more radical methods.

The key idea of our approach is to dynamically identify the speculative cache lines that create conflicts, and to detect conflicts using these lines only. We manage non-conflicting lines privately to processor cores, without generating network traffic. To guarantee that all conflicts are detected, we associate a very small amount of metadata (2 bits) with non-conflicting lines. If the metadata indicates that an access to a line may create a conflict, a simple hardware logic builds the full conflict detection information from the list of current line sharers, and precisely detects the conflicts.

Figure 6.1 quantifies the dominance of non-conflicting cache lines in current TM workloads. For simulator configurations between 1 and 32 processor cores, we executed all applications from the STAMP benchmark suite. The *horizontal axis* shows the name of an application, the number of simulated processor cores, and their number of non-conflicting lines. The *vertical axis* shows the maximum (not the cumulative) number of conflicting lines during the execution of an application. We count the number of conflicting lines when a transaction commits and aborts.

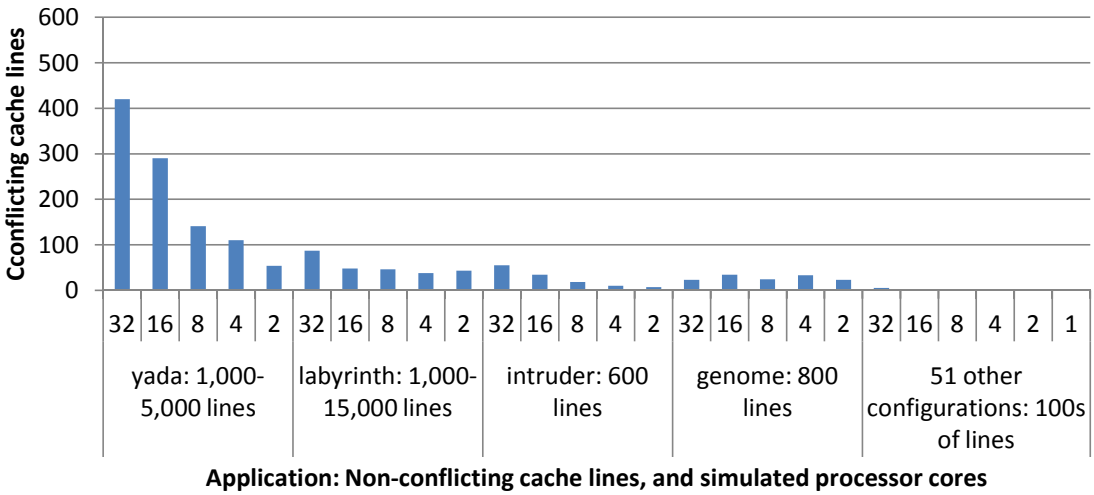


Figure 6.1: **Very few cache lines create genuine conflicts in existing TM workloads**, an observation weakly exploited in previous HTM proposals. The vertical axis shows the number of conflicting lines, and the horizontal axis shows the workloads and the number of non-conflicting lines. We show the maximum (not the cumulative) number of transactional cache lines in each application.

These results indicate that, although existing HTM proposals must support up to 15,000 transactional lines at some points of execution (the case of labyrinth), only 10s to 100s of lines actually determine the conflicts and *need* to be analyzed to guarantee correct execution. The conflict detection using non-conflicting lines generates only wasted work, time, and energy.

In Section 6.2, we introduce EcoTM (**Eco**nomical**TM**), an HTM that leverages the separation of conflicting from non-conflicting cache lines. EcoTM identifies the conflicting cache lines based on run-time accesses. It works dynamically, automatically, and transparently to the programmer. Even if a conflict occurs on a cache line that was non-conflicting until that time, EcoTM recovers and detects the conflict correctly.

After that, we explain the mechanism for identifying conflicting lines. In this work, we assume the common notion of conflicting cache line: a line that is read and written by two or more concurrent transactions. In other words, a cache line is conflicting if: a transaction writes to a line, while other concurrent transactions access the same line. On the other side, a line is non-conflicting if it is: (1) read-only, accessed by 1 or more concurrent transactions, (2) read-write, accessed by a single transaction, or (3) not accessed concurrently by transactions.

In the same section, we describe how EcoTM detects conflicts eagerly using only the lines it identifies as conflicting, then how EcoTM marks conflicts in a dedicated hardware structure, and finally how EcoTM resolves conflicts to provide correct execution, by aborting only genuinely conflicting transactions.

The basic EcoTM architecture targets a Chip-Multi-Processor (CMP) system with MESI (or similar) cache coherency protocol and with memory directory. The basic EcoTM handles only transactions that fit into private caches (like EazyHTM). In Section 6.3, we describe how EcoTM can be extended to support transactions larger than private caches (overflowed transactions), and to support commodity CMP systems that have limited size of directory (e.g., a directory only in the shared L2 caches).

EcoTM manages non-conflicting lines locally to processor cores, and generates conflict-detection traffic only for the few conflicting lines. This reduces the conflict detection traffic in the system, and increases the efficiency and performance of HTM, without impacting the correctness. We evaluate the performance of EcoTM, and analyze the sensitivity of EcoTM performance, and we show the results of our evaluation in Section 6.4.

6.2 Basic EcoTM Architecture

In this section, we outline the baseline CMP architecture, including the EcoTM extensions to processor cores, private caches, and directory. We start by describing EcoTM architecture at high level, then explain how non-conflicting transactions execute (and commit and abort) locally to processor cores, and continue by explaining the mechanism for identifying conflicting lines. After that, we illustrate EcoTM execution with several execution scenarios.

The basic functionality of EcoTM is the same as with EazyHTM. EcoTM detects and marks conflicts eagerly, during transaction execution, but postpones the resolution of conflicts until a transaction tries to commit the speculative changes. At this point, the committing transaction has a prepared list of conflicts and transaction commit can be fast and simple. This provides high concurrency and reduces the amount of work that needs to be done when transaction commits. In result, eager-lazy HTMs have the benefits of both eager and lazy conflict management:

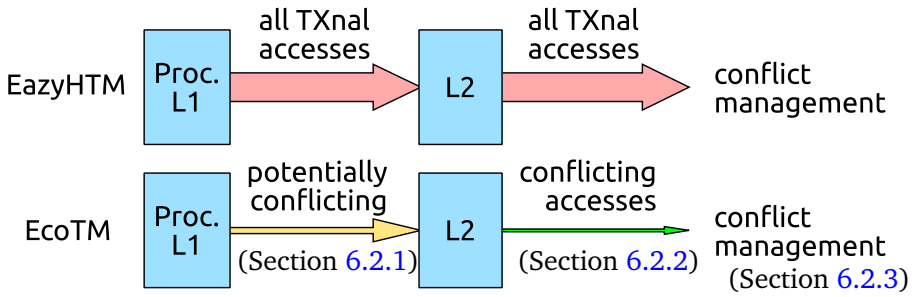


Figure 6.2: An overview of the conflict detection in EcoTM. Exclusive lines are handled in private L1 caches. Shared lines are forwarded to L2 cache, which identifies the conflicting lines and answers the non-conflicting requests (Section 6.2.2). The conflict detection is done only for the genuine conflicting lines.

(1) just as lazy, they have good scalability for high-contention workloads, and (2) just as eager, they have simple transaction commits.

Since eager-lazy HTMs detect conflicts eagerly (during transaction execution), their bandwidth requirements are lower than for pure-lazy HTMs, especially during commit operation. In addition, eager-lazy HTMs have better overall performance than pure-lazy HTMs. However, compared to pure-lazy HTMs, eager-lazy HTMs may have higher traffic during regular transaction execution.

EcoTM focuses on optimizing and reducing the traffic in the EazyHTM protocol, when it executes regular (L1-bounded) transactions. We illustrate the optimizations in Figure 6.2, and we will mention them individually in the following sections. Higher traffic may (1) reduce the performance if a system has limited-bandwidth ICN, or (2) increase the energy consumption.

Figure 6.3 shows the baseline architecture for EcoTM. To support EcoTM, processor cores have the following extensions: (1) backup and restore mechanism for the register file, (2) speculative flags (read, write, and conflicting) associated with L1 cache lines, (3) logging support for overflowed transactions (explained in Section 6.3), and (4) transaction state, which is set during execution and reset when transaction aborts or commits. A transaction state denotes a transaction to be: (1) active or inactive, (2) conflicting or non-conflicting, and (3) local or overflowed.

A transaction becomes *active* when it begins execution, and *inactive* when it terminates (either commits or aborts). A transaction begins as *non-conflicting*, and changes to *conflicting* if a conflict is detected with other transactions in the system. Finally, a transaction begins as *local*, meaning that it fits and executes in L1 cache.

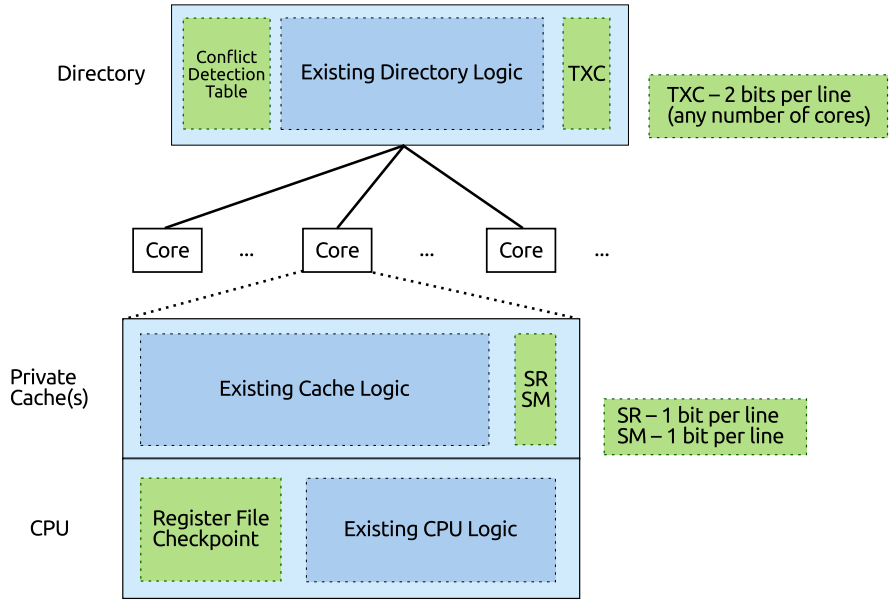


Figure 6.3: The baseline Chip-Multi-Processor (CMP) architecture for EcoTM. The architecture of a single core is similar to the one in EazyHTM. The directory has an additional Conflict-Detection Table, and a 2-bit metadata in directory entries to identify the conflicting cache lines.

If a cache lines touched by a transaction needs to be evicted from the L1 cache, the transaction is marked as *overflowed* and continues execution. We give more details on the execution of overflowed transactions in Section 6.3.

6.2.1 Core-local transactions

A large class of transactions can execute locally to processor cores – for example, read-only transactions, or transactions that have all cache lines exclusive in a processor core. Read-only and exclusive cache lines cannot create conflicts, and they do not generate off-core traffic (outside of L1 caches). This allows EcoTM to efficiently support the privatization idiom [49, 67, 72], and the compiler- or programmer-defined thread-local variables [80].

To detect exclusive or read-only lines, EcoTM relies on the line state assigned by the MESI protocol in the directory. MESI protocol already assigns exclusive ownership to a line accessed only by one L1 cache. Similarly, MESI protocol already assigns a shared state to a line accessed by more than one L1 cache.

If, during transaction execution, a line changes the state from exclusive or read-

only to read-write, EcoTM protocol notifies current line sharers, resulting in correct conflict detection. After that, line is marked as conflicting, which forces conflict detection for future line accesses.

An access to a cache line that is neither exclusive nor read-only generates an off-core traffic (traffic outside of L1 cache). If the access does not result in a conflict, a transaction can abort or commit core-locally, without communicating with other concurrent transactions. A non-conflicting transaction terminates by first clearing the speculative flags in L1 cache, and then restoring (for abort) or flushing (for commit) the checkpoint of the register file, created when transaction began execution.

6.2.2 Identifying Conflicting Cache Lines

In this section, we describe how EcoTM identifies conflicting cache lines, without explaining the mechanism for conflict detection and resolution. The majority of cache lines never create conflicts between transactions, but it is not easy keep track of this, due to the volatile nature of conflicts. A cache line may become conflicting during the execution of some transaction, and after that a line may again become non-conflicting.

The design of EcoTM allows rapid changes between the conflicting and the non-conflicting states of cache lines. The directory identifies a conflicting line based on the list of sharers (already existing in directory-based systems) and the state of a (EcoTM-specific) metadata associated with a cache line.

The metadata associated with cache lines has only 2-bits, and we name it a Quick Conflict Check (QCC) state. The list of line sharers and the QCC state are logically associated with a cache line while, physically, they are a part of the tag of a directory entry. We assume that directory entries are distributed, and accessible through the banked L2 cache.

The two bits of QCC encode the following states:

1. NonTX: non-transactional line (not accessed by any transaction)
2. TReadonly: read-only by one or more transactions,
3. TExclusive: exclusive to a transaction (may be written), and

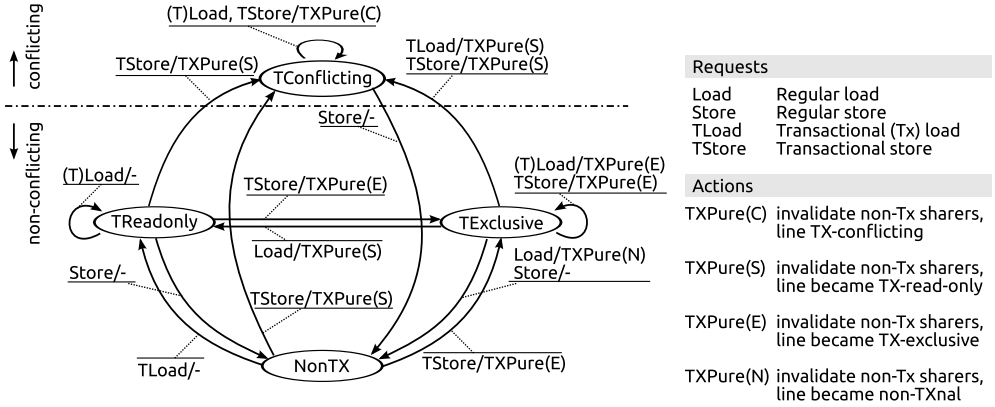


Figure 6.4: State-transition diagram of Quick Conflict Check (QCC), which identifies the conflicting lines (directory entries).

4. TConflicting: conflicting line (read *and* write by more than one transaction).

We show the state diagram of the QCC states in Figure 6.4. A cache line is initially non-transactional (QCC:NonTX). After a first transactional load (TLoad), QCC state transitions to QCC:TReadonly, and after transactional store (TStore) to QCC:TExclusive. A line becomes conflicting (QCC:TConflicting) if it has both TLoad(s) and TStore(s) from different processor cores. The QCC state is reset if there is a regular (non-speculative) write to a line.

If a line is exclusive to the L1 cache, it cannot create conflicts between transactions. Therefore, for lines in exclusive mode (i.e., Exclusive, Owned, or Modified) neither TLoad nor TStore are sent. If a line is shared between L1 caches (processor cores), transactions detect conflicts by sending TLoad and TStore requests to the L2 cache.

TStore is sent instead of a regular Store request, if a line is not exclusive to the L1 cache. If a directory receives a TStore, it always executes a TXPure action. Conversely, TLoad is sent lazily, as a response to a TXPure action. In case L1 cache does not already have the line when it should transactionally read a location, it issues a regular Load request. If the requested line is in QCC:TConflicting or QCC:TExclusive state, the directory initiates the TXPure action.

L2 cache executes a TXPure action when a line may become conflicting, or when a new conflict may be marked. This happens, for example, when a line is shared and there has been a previous TStore to the line. TXPure evicts a line from

all non-transactional sharers of the line, which may avoid some spurious changes to QCC:TConflicting. For example, after a TXPure action a line may become exclusive to a transaction, resulting in QCC:TExclusive instead of QCC:TConflicting. If line state changes to QCC:TConflicting, EcoTM protocol initiates conflict detection described in Section 6.2.3.

Lazy QCC updates. L2 cache (the directory) may have stale information, since it is not informed of the TLoads in L1 cache. Therefore, L2 *always* consults L1 cache before marking a declaring a line as conflicting (QCC:TConflicting), or before marking a conflict between transactions.

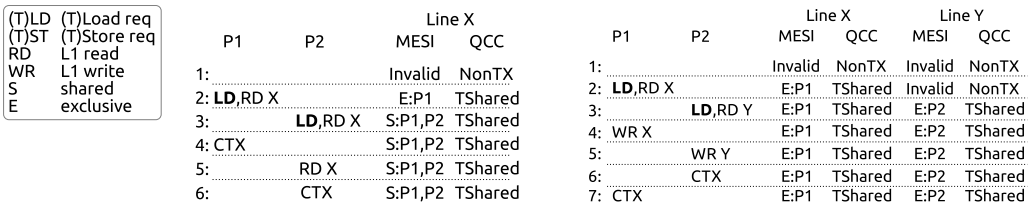
L2 cache executes a **TXPure** action, which invalidates all non-transactional line sharers. TXPure invalidates the line copies in L1 caches which are not a part of a transaction (speculative flags set for the line). The TXPure invalidations are similar to the invalidations in a regular "Store" request. After updating the list of line sharers, the directory updates the QCC state. If there is more than one transactional sharer after the TXPure action, a line is marked as conflicting (QCC:TConflicting).

When a transaction terminates (aborts or commits), QCC state is cleaned lazily. This makes commits fast, and leaves the task of cleaning stale QCC state to future transactions. Lazy updating of QCC reduces the amount of L1-L2 traffic, the latency of operations, and the number of QCC state changes, while it still guarantees that no conflicts will be missed.

Running examples of core-local transactions. Figure 6.5a demonstrates a running example of two read-only transactions executing on processor cores P1 and P2. A processor communicates with the directory for the first speculative read, by sending a regular Load request. For subsequent reads from the same variable, a processor would access only the L1 cache. After committing, the processor cores do not clear the QCC state.

Figure 6.5b demonstrates a different scenario, in which two cores acquire exclusive access to two different cache lines. In this case, a processor speculatively writes to the L1 cache without notifying the directory. To commit, a processor core locally (in L1 cache) upgrades the speculatively modified lines to the regular Modified state, and continues the execution.

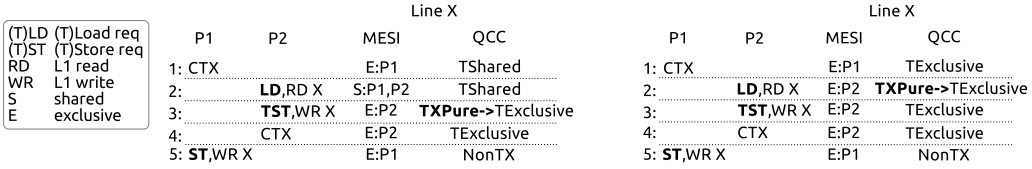
In the two previous examples, the EcoTM protocol does not increase the traffic on the interconnects, compared to a regular non-transactional execution.



(a) Speculative reads generate regular Load requests.

(b) TStore requests are not sent, if L1 cache has the exclusive access to a line.

Figure 6.5: Core-local execution of the most common, non-conflicting transactions in EcoTM. The off-core traffic (bold font) is not increased by executing these transactions.



(a) Lazy update of the QCC:TReadonly state.

(b) Lazy QCC:TExclusive update.

Figure 6.6: Lazy updating of QCC states permits fast core-local commits and aborts, without introducing false conflicts.

Running examples of lazy QCC updating. Figure 6.6a illustrates the lazy updating of a stale QCC:TReadonly state, left after P1 committed a transaction (step 1). P2 issues a Load request (step 2), and after that a potentially-conflicting TStore request (step 3). Before marking a line as conflicting (QCC:TConflicting), L2 cache invokes a TXPure action. TXPure invalidates a copy in P1, since P1 does not access the line speculatively, and this makes P2 an exclusive owner of the line. Therefore, instead of become conflicting, the line changes to QCC:TExclusive. The QCC state is left unchanged when transaction commits (step 4). Later in execution, P1 makes a regular Store to the line (step 5), which resets the QCC state to QCC:NonTX.

Figure 6.6b shows an example of a stale QCC:TExclusive state, left after P1 commits (step 1). Incoming Load from P2 (step 2) initiates a TXPure action, which invalidates a copy in P1, since P1 does speculatively access the line. This makes P2 an exclusive sharer of the line (step 3). The QCC state is reset to QCC:NonTX when P1 makes a regular Store to the line (step 5).

6.2.3 Conflict detection and resolution

If QCC gets into the conflicting state (QCC:TConflicting), a line is forwarded to a dedicated hardware logic, which precisely detects conflicts, and arbitrates the commits of the conflicting transactions. The dedicated hardware logic is simple, with complexity comparable with a "get commit sequence number", often proposed for lazy HTMs. This section describes this mechanism for conflict detection and management.

When QCC identifies a conflicting cache line, it informs the requesting processor core that its current transaction is conflicting. The processor core locally changes the transaction state from non-conflicting to conflicting, and the conflict is marked in the directory, in a structure called conflict bitmap.

A transaction with a conflicting execution cannot commit locally any more. It has to requests a commit permission from the conflict bitmap, and has to wait for a response before it proceeds with the commit. Conflict bitmap has simple functionality and responds fast and in constant time. On a commit request, conflict bitmap checks if the requesting transaction is still valid. If it is not, the commit request is rejected. If the committing transaction is valid, the conflict bitmap invalidates all conflicting transactions of the committing transaction by sending them an aborts request. The cores cannot reject the abort request from the conflict bitmap, so the bitmap immediately sends an acknowledgement to the core that asked for the commit permission.

In processor core, commit behaves similarly to a memory fence [41, 50, 78]. That is, a commit allows the memory operations to be reordered before the commit, but forces all loads and stores before the commit to be ordered with respect to the loads and stores after the commit. The processor core stops writing-back (retiring) instructions [71] that follow the commit, until the commit operation is complete.

After commit, an L1 cache lazily write backs the committed data when: (1) it evicts the line (due to the capacity constraint), (2) it reduces the access mode to shared or invalid (for example, other core requests line access), or (3) it speculatively modifies the line in a new transaction.

If a transaction with a conflicting execution aborts, it informs the conflict bitmap. The conflict bitmap clears all existing conflicts with the aborted transaction.

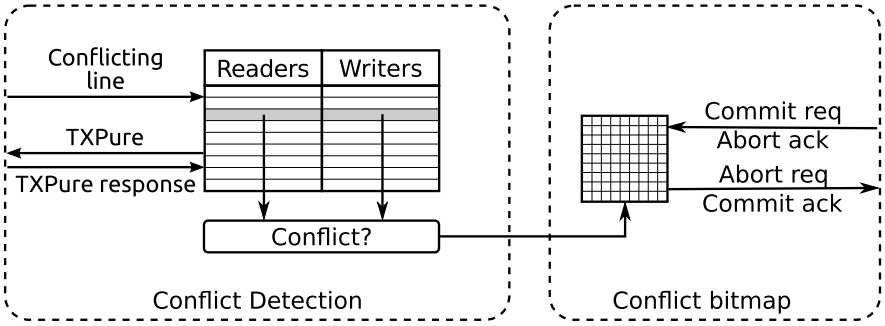


Figure 6.7: Conflict management hardware in EcoTM. Conflict detection table handles only conflicting lines. It detects conflicts precisely, and marks them in the Conflict bitmap.

Conflict-Detection Table. Figure 6.7 outlines the hardware extensions for conflict management. Conflict detection table has a limited number of entries, and handles only conflicting lines. An entry in the table has a bitmap of speculative accesses: 1 bit to mark a Speculative Read (SR) and 1 to mark a Speculative Write (SW) of each processor core. In a 32-core system, each entry in the table has 64 bits. Since a table entry has a complete list of speculative accesses, conflicts can be detected precisely, just as a naive lazy HTM would do: a speculative write bit marks a conflict with all other line accessors with SR or SW bit set.

QCC forwards all accesses to a conflicting line to the conflict-detection table. A new entry in the table is initialized from the existing list of line sharers. The L2 cache executes a TXPure action, which leaves only speculative accessors in the list of sharers, and the L1 caches also return the type of their speculative access (read or write).

Since conflict-detection table has limited number of entries, an entry may be invalidated at any time. When a conflicting line is re-accessed, the entry in the table is re-built the same way as the first time, by executing the TXPure action.

A TXPure action sends a multi-cast message to all line sharers (processor cores), just as a regular Store request. To reduce the number of the multi-cast request, it is important to have a sufficient number entries in the conflict-detection table.

Conflict bitmap. A conflict detected by the conflict-detection table is marked in the conflict bitmap. Conflict bitmap is a $(N-1)^2$ bit-matrix (N is the number of processor cores) of conflicts between transactions. In it, each core has one bit-vector, which presents the transactional conflicts between this core and all other

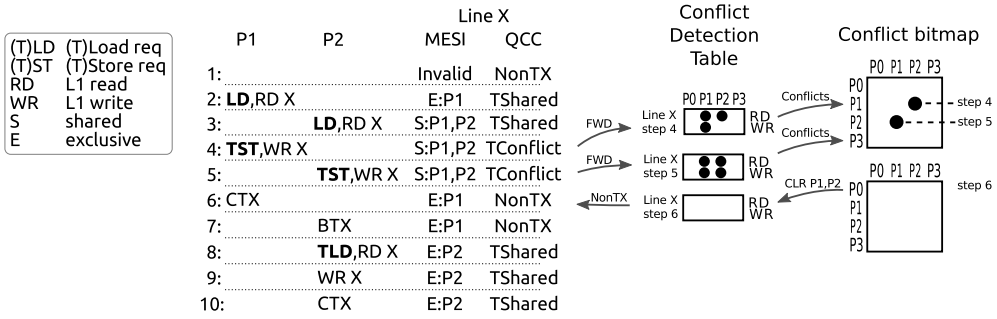


Figure 6.8: Execution of conflicting bounded transactions. The conflicts from P1 to P2 and from P2 to P1 are precisely detected in the conflict-detection table, and marked in the conflict bitmap. When P1 requests to commit, conflict bitmap aborts the transaction in P2.

cores. For example, if a bit-vector of core 1 has set bit 2, this means that if transaction 1 wants to commit, it needs to abort the transaction 2.

When conflicting transaction commits or aborts, it informs the conflict bitmap. This flash-clears the transaction column in the conflict-detection table, and all entries for the transaction in conflict bitmap.

6.2.4 Example of conflict management

Figure 6.8 illustrates an example of a conflicting execution of transactions on processor cores P1 and P2. The two transactions speculatively modify the same cache line. QCC state identifies a line as conflicting (step 4), when P1 speculatively writes to the line. An entry in the conflict-detection table is initialized, a conflict from P1 to P2 is detected and marked in the conflict bitmap.

A second conflict (step 5) occurs when P2 also speculatively writes to the line. An entry in the conflict-detection table is updated, and a conflict from P2 to P1 is marked in the conflict bitmap. When P1 commits (step 6), conflict bitmap requests an abort from P2, and flash-clears the P1 and P2 columns in the conflict-detection table. The cache line is no longer transactional, and the line state is updates to QCC:NonTX. After that, P2 re-executes the transaction (steps 7-10), this time without any conflicts.

6.3 Overflowed transactions

In this Section, we describe the handling of overflowed transactions, and the extending of the EcoTM mechanism to the commodity CMP systems with limited directories (for example, to a CMP with a directory only in L2 caches).

6.3.1 Conflict management for overflowed transactions

The conflict management in EcoTM is generally similar between the core-local and the overflowed transactions. In this section we explain the subtle changes in the coherence protocol, that enable correct conflict detection, even with overflowed transaction.

Silent L1 evictions. Even if a speculative line is evicted from L1 cache, the directory should not delete the information about the line access. That is, the directory should believe that the L1 cache still has the line. For this, the EcoTM protocol relies on silent evictions for transactional lines [55]. Silent evictions are already performed by conventional MESI protocols to reduce the bandwidth overhead, by silently evicting exclusive and shared lines. Thus, if a speculative line gets evicted from L1 cache, the directory information is not updated to reflect the eviction.

Conservative line sharing. For the core-local transactions, the QCC changes states based on the speculative flags in L1 caches. For the overflowed transactions, we have to be conservative and to always assume that a core marked as line sharer also speculatively accessed it. In particular, for QCC:TReadonly state we assume that all line sharers transactionally read the line. For QCC:TExclusive or QCC:TConflicting state, we assume that all line sharers transactionally read and modified the line.

This obviously introduces some false conflicts, but the probability of these false conflicts is low, with a sufficient number of entries in the conflict-detection table. If a related entry in the table does not get evicted, no false conflicts are introduced even if a line is evicted from the L1 cache. In Section 6.4 we sensitivity of EcoTM performance to the number of entries in conflict-detection table.

6.3.2 Logging QCC changes

Most QCC changes converge to the accurate state, when regular Store requests reset the QCC state or when TXPure actions update the QCC state. However, if a line has overflowed sharers, stale QCC:TExclusive or QCC:TConflicting may indicate a conflict that does not actually exist. Conflict-detection table offers a solution, but the entries in this table can be silently evicted.

To address this, a processor core creates a QCC undo-log for the overflowed lines with QCC:TExclusive or QCC:TConflicting state. To facilitate the processing of the entries from the logs, each QCC state has a separate log. The logs are organized as stacks and are stored in a cacheable thread-private memory. The hardware pushes the addresses, and a software handler removes the entries and resizes the log if necessary. Note that the logs contain only overflowed lines, which makes it unlikely to repeat entries.

EcoTM logs have the hardware and software requirements similar to the LogTM [55] logs. The advantages over LogTM-style logs are that: (1) EcoTM logs only for overflowed lines, and (2) EcoTM does not log any data. EcoTM logs only addresses, and this significantly increases the efficiency of logs. EcoTM needs only 8 bytes per entry (for line address), whereas a LogTM-style log needs 64+8 bytes (line data+address) per entry. LogTM needs to log the data as well, since it restores the original, non-speculative values when a transaction aborts. In contrast, in EcoTM the non-speculative values are already in shared memory, and to abort a transaction EcoTM only flushes the private speculative data of the transaction.

6.3.3 Data management for overflowed transactions

EcoTM mechanism generally supports any overflowed-data-management mechanism used by either lazy, or eager-lazy HTMs. Therefore, this work assumes the that other mechanisms for managing overflowed data exist. We name such support an Overflow Buffer (OB), and assume that it is organized similarly to the one proposed by Shiraman et al. [69]. The OB is organized as a simple per-thread hash table in virtual memory, and accessed by the OB controller that sits on the private cache miss path. On private cache misses, the request is redirected to the OB and handled in hardware. The commit-time write-backs are performed by the

OB controller, and occur in parallel with other useful work by the processor.

In Section 6.4 we analyze the dependence of EcoTM performance from the OB latency, and show that the OB latency does not significantly affect the EcoTM execution time.

6.3.4 Support for context switching and interrupts

The transactions are usually much shorter from the time between interrupts, which commonly trigger contexts switches. The transactions in existing workloads do not execute longer than several thousand cycles, and the context switches happen every tens to hundreds of millions of cycles on the mainstream microprocessors (the default interval for context switches is $10ms$).

Eager HTMs generally have slow transaction aborts, and the abort overheads can easily exceed the overheads of a transaction migration, even if the migration process is complex. In contrast, EcoTM has fast aborts. When an aborted transaction starts executing again on a different processor core, it will likely finish in several thousand cycles. The complete execution can be faster from many proposed transaction migration mechanisms.

While future workloads might call for a different approach, enabling migration of long-running transactions is likely to be extremely complex, particularly in systems combining the use of operating systems and virtual machine monitors.

Despite our position on supporting context switches and the transaction migration, EcoTM can relatively simply be extended to support them. One way could be by saving the speculative state before a context switch, and re-applying it after the context switch, on a different processor core.

6.3.5 EcoTM on Systems with Limited Directory Size

On multiprocessor systems with a limited directory size, for example, with a directory only in L2 or L3 caches and not in the entire physical memory, the history of speculative accesses could be lost once a line is evicted from the top-level cache. This could cause future speculative accesses to the line to miss real conflicts.

To prevent this from happening, some amount of directory data can be saved

in physical memory together with an evicted cache block. We show two configurations: (1) saving a complete list of cache line sharers together with QCC, or (2) saving only the QCC (2 bits per cache block). Saving less metadata in memory reduces the storage requirements, but risks false conflicts in future execution.

The conflict-detection metadata can be preserved either in the ECC area (by switching to SECDEC code), as has been proposed by TokenTM [17], or by increasing the number of bits in DRAM rows. The DRAM row size already increases between different generations. The saved EcoTM metadata can be stored as regular data in DRAM, since it does not require any additional logic.

6.4 Evaluation

We evaluated EcoTM using M5, a full-system simulator of the Alpha architecture [11]. We replaced the default bus based cache coherency with a MESI based directory cache coherency.

Beside EcoTM, we implemented (i) LogTM-SE [79], (ii) an unbounded eager HTM with perfect signatures, and (iii) an ideal-lazy HTM. We ported the LogTM-SE code to M5, from its original publicly available source code (implemented with Simics and GEMS), and verified that the performance is comparable with the original. LogTM-SE is a state-of-the-art unbounded HTM proposal, widely used by other unbounded HTM proposals as the base HTM. It is therefore an excellent reference point for comparing EcoTM with other proposals.

In all HTM configurations, we use in-order cores with a fixed 1 CPI for non-memory related instructions. The memory operations take 1 cycle plus a variable latency returned by the memory subsystem. An overview of the underlying hardware and the latencies is given in Table 6.1.

We evaluate the proposals using STAMP TM benchmark suite. Detailed benchmark characteristics are given in [20].

Execution time breakdown. In Figure 6.9, we show the breakdown of the total execution time for EcoTM executing all STAMP applications over 1–32 processor cores. The total execution time is split into four categories: (1) *Thread start/end* – the time spent in thread synchronization during entering and leaving parallel sections, (2) *Quiescent* – the time spent in quiescent state, which usually occurs

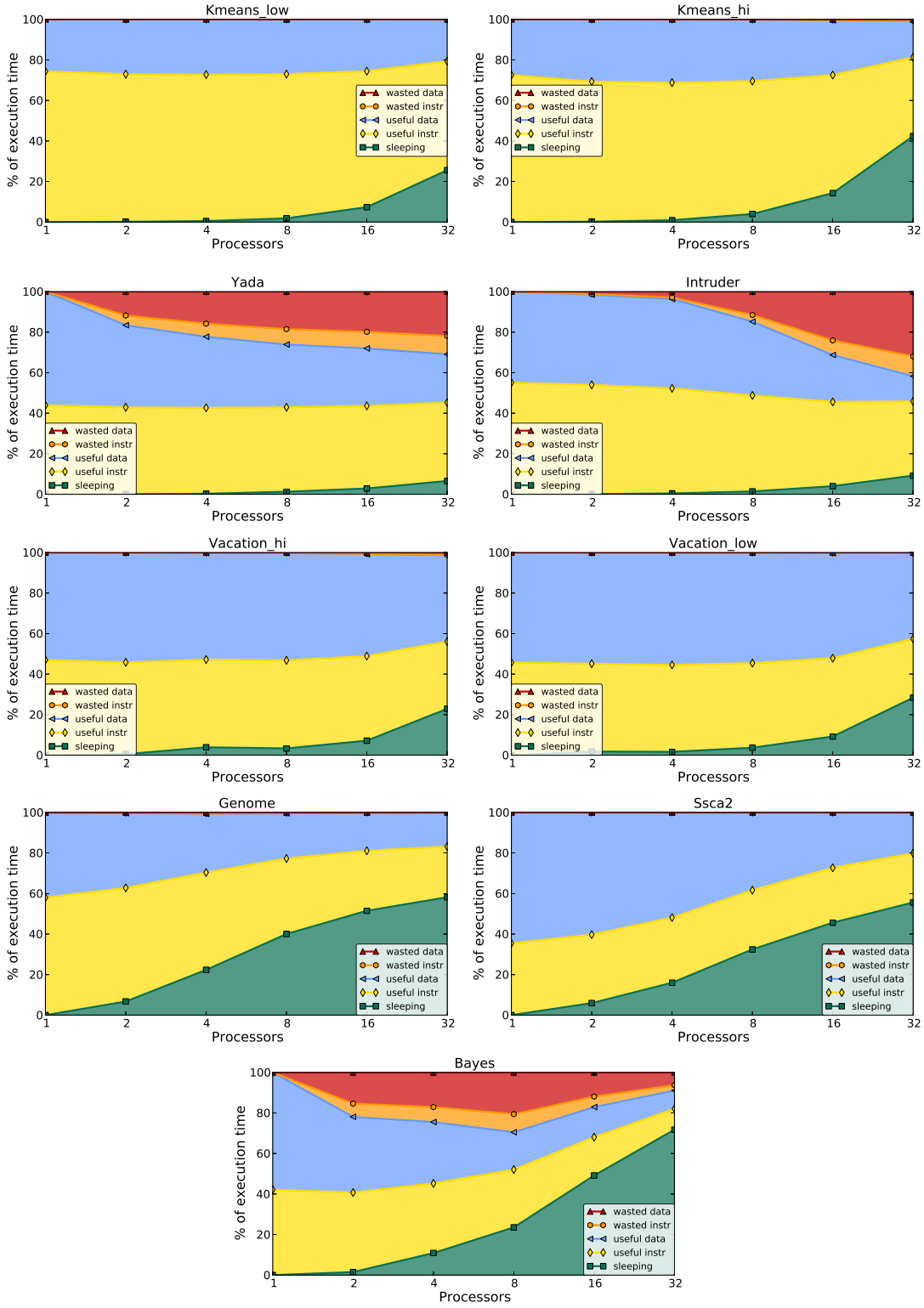


Figure 6.9: A breakdown of the total execution time in STAMP applications, for 1-32 processor cores

Processor	1-32 cores, single-issue, single-threaded, 1 CPI
L1 Cache	32KB private, 2-way, 64-byte blocks, 2 cycles, write-back
L2 Cache	8MB banked NUCA, 8-way, 64-byte blocks, 32 cycles, write-back
Directory	Bit-vector of sharers, 8-cycle access latency
Memory	4GB, 500 cycles latency
Interconnect	2D Mesh, 4-cycle latency
Conflict-detection table	256 directly mapped entries (unless otherwise noted)
Conflict bitmap	40-cycle access latency
Log latency	8-cycle latency for removing one entry

Table 6.1: The hardware configuration

after being unable to enter a barrier after several successive retries, (3) *Useful* – the time spent outside of transactions, and in transactional code that successfully commits, and (4) *Wasted* – the time spent in transactional code that is rolled back due to abort.

The breakdown indicates problems with the parallelization of several STAMP applications. First, Bayes has unbalanced work between threads, since it spends significant time synchronizing the threads the start and the end of the workload execution. Kmeans and Vacation have a similar problem, but to a smaller extension. Second, Genome and SSCA2 rely on barriers for synchronizing thread progress, which is seen by the significant time in the quiescent (idle) processor state during the execution. The mentioned non-executing time in these applications consumes over 50% of the execution time with 32 cores, and therefore the application performance does not correctly reflect the HTM performance.

Wasted work becomes a significant factor for applications with medium and high contention, Bayes, Intruder, Labyrinth, and Yada, especially with higher number of execution threads.

Performance and Scalability. Figure 6.10 presents an evaluation of speedup over the sequential execution of the same application (that does not use threads or locks).

Our evaluation includes four HTMs. The first HTM is the *ideal-lazy* unbounded HTM, a lazy conflict resolution HTM with no latencies (all operations are instant and without overheads). All speculative reads and writes are simple cache reads, and no transactional messages are sent during execution. When transaction commits, it magically detects and resolve conflicts with other transactions and then

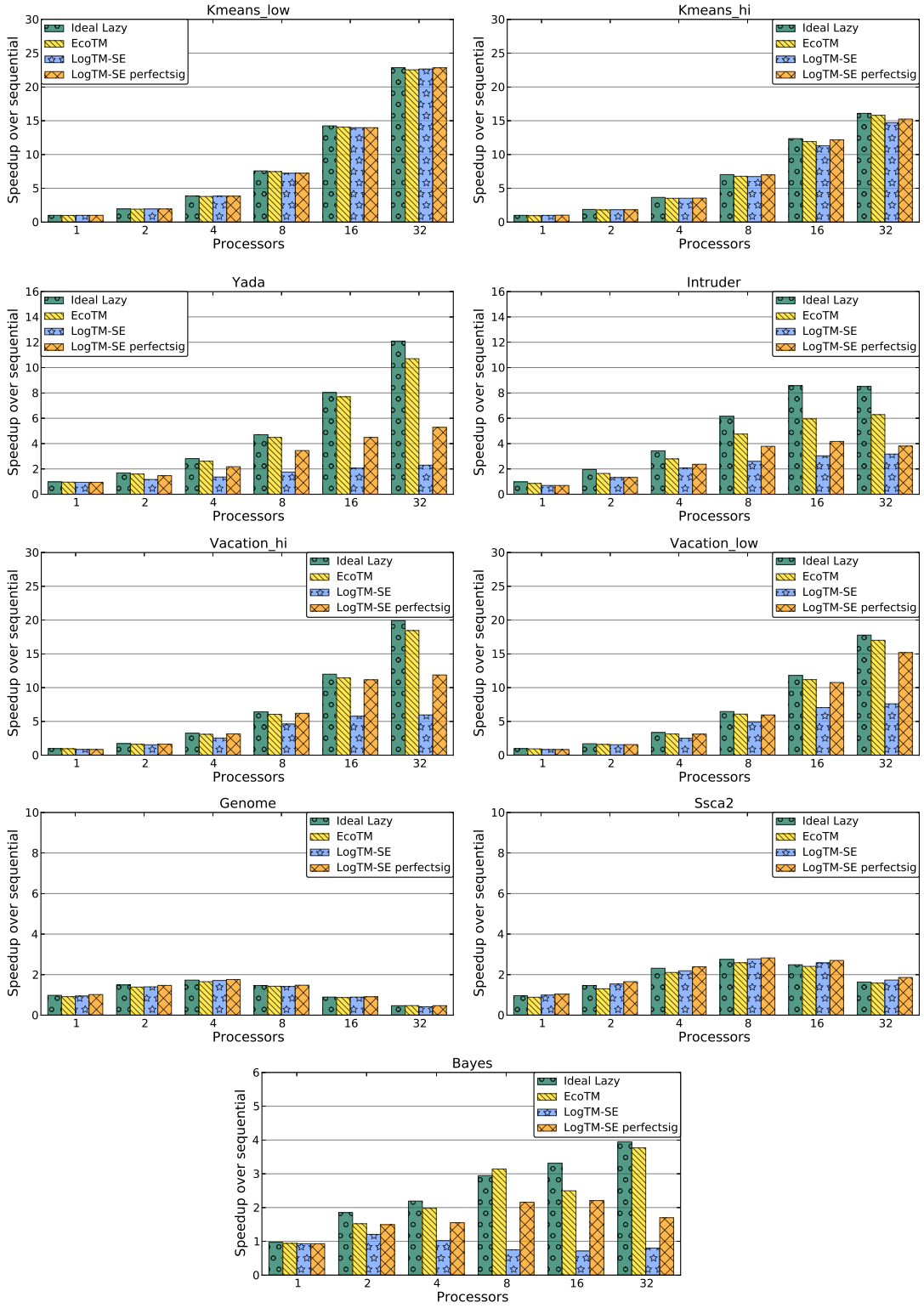


Figure 6.10: The speedup of the STAMP TM benchmark suite applications normalized to the sequential execution (no threads or locks). The overflow buffer latency is fixed to 100 cycles for EcoTM, and 0 cycles for the ideal-lazy HTM.

publishes the speculative changes. There are no false conflicts except due to false line sharing. No data moves during publishing modified lines. All speculatively-modified lines are magically converted into modified state, even if they got evicted from private caches. The overflow buffer has zero latency and infinite capacity. Second is *EcoTM*, and the third is *LogTM-SE*, which we configure with exponential backoff and 2-Kbit Bloom-filter signatures with two parallel hash functions, as proposed by the LogTM-SE authors. The last is the *eager-perfect* unbounded HTM, based on LogTM-SE. It has perfect signatures, that do not create any false conflicts. TokenTM authors report a performance comparable with this HTM.

In our evaluation, lazy conflict management has better performance than eager if: (1) the contention is high, or (2) the transactions are large. With small transactions, eager and lazy conflict management have similar performance. Eager would likely have better performance with large transactions and low contention. However, STAMP does not include such application.

Overall, EcoTM performance is close to the performance of ideal-lazy HTM, which bounds the performance of unbounded HTM. EcoTM has better performance in all sets of executions from both LogTM-SE and eager-perfect HTM. The geometric mean of the performance improvement of EcoTM for all STAMP applications is 35.7% over LogTM-SE with realistic Bloom filter signatures, and 8.8% over eager-perfect HTM.

Genome and SSCA2 almost do not scale in any of the implemented HTMs. This happens because of the inefficient implementation of barriers on this architecture. Since these applications have small transactions, and as can be seen in Figure 6.9, that the amount of wasted work (transaction aborts) is minimal in these applications, the reason for bad scalability is not in HTMs.

Especially important in our evaluation are Bayes, Labyrinth, and Yada, which have large transactions, that frequently overflow the small L1 caches from our simulator.

Bayes in our evaluation shows similar behavior to the one presented by the STAMP authors. Since the work is not well balanced between threads, the execution time of Bayes does not necessarily represent the HTM performance. Labyrinth was excluded from this figure, since it depends on early release [20], which our LogTM-SE implementation does not support.

Yada has very large transactions and moderate contention, which makes it a good application for evaluating the deficiencies of unbounded HTMs. A 1-32 thread geometric mean of the EcoTM performance is 5% from the ideal-lazy HTM, and 24% better than the eager-perfect HTM. With 32 threads in particular, EcoTM finishes 10.66 times faster than the sequential code, which is only 11.4% slower than the ideal-lazy HTM. With the same configuration, eager-perfect HTMs finishes 5.3 times faster than the sequential code, but 2 times slower than the ideal-lazy HTM. Similar performance difference between eager and lazy HTMs for this application was also reported by the STAMP authors.

Over all STAMP configurations, a geometric mean of the difference between EcoTM and ideal-lazy HTM is 7.1%.

A weak implementation of Bloom-filter signatures in our LogTM-SE introduces many false conflicts, and significantly hurts its performance. Note that, while a better signature implementation (for example, using H3 or PBX hash functions) would improve the performance of LogTM-SE, the performance of such HTM would still be below the ideal-perfect HTM.

EcoTM overheads. EcoTM overheads come from: (1) logging, and (2) conflict management. We evaluate the EcoTM overheads by comparing the execution with a no-overhead, ideal implementation. EcoTM overall stands very close to the ideal-lazy HTM. The biggest difference is observed for the Intruder, where the 32-core ideal-lazy HTM is approximately 40% faster than EcoTM. Where ideal-lazy HTM has instantaneous aborts, EcoTM needs to manage logs and negotiate the commit or abort with the conflict bitmap. Excluding Intruder, EcoTM is within 5.2% from the ideal-lazy HTM on average for all configurations, and within 5.6% and 4.2, respectively, for 32-core executions.

Sensitivity to the size of the conflict-detection table. Adding more entries to, and increasing the associativity of conflict-detection table: (1) reduces the number of false conflicts introduced by overflowed transactions, and (2) reduces the traffic on the interconnects. While the former directly affects the execution time, the latter affects the power consumption. Figure 6.11 shows the sensitivity of the EcoTM performance to the number of entries in the table. All STAMP applications except Yada have the same performance even with only 8 directly mapped entries. Yada is more sensitive than the rest. However, only 256 directly mapped entries

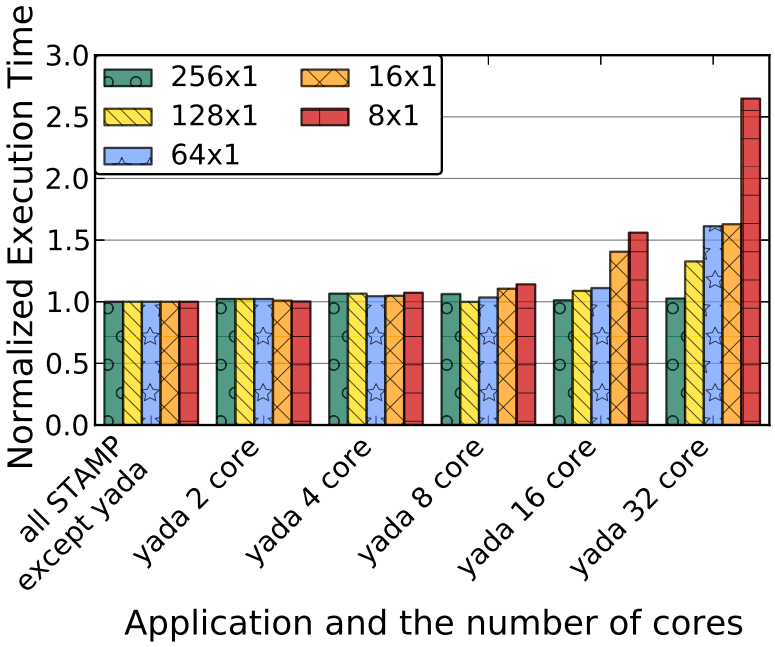


Figure 6.11: Even small conflict-detection table works well in almost all configurations. The execution time is normalized to the unbounded fully-associative table. Even a directly mapped 16-entry (1-way) conflict-detection table provides good results for current workloads.

are enough for avoiding false conflicts.

Impact of the overflow buffer (OB) latency. The OB stores the speculatively-modified lines that overflow the private cache. In Figure 6.12, we show how the speed of the OB affects the performance of EcoTM. The presented results are for the OB latencies of 10 and 100 cycles per access. The execution times of EcoTM is normalized to the configuration with the 0 cycles per OB access, which would be the ideal OB. We see that Bayes has slightly better performance when OB access has some latency. Since Bayes has non-deterministic (work-stealing) execution, we cannot conclude anything with these results. One more result is interesting, Intruder, where performance also improves when OB has some latency. Since Intruder has a highly-conflicting execution, reducing the speed of transaction execution results in less wasted work and the overall performance of the application reduces. We can conclude that, with the current workloads, the speed of the OB does not influence the execution time as much as the choice of the policy for conflict management.

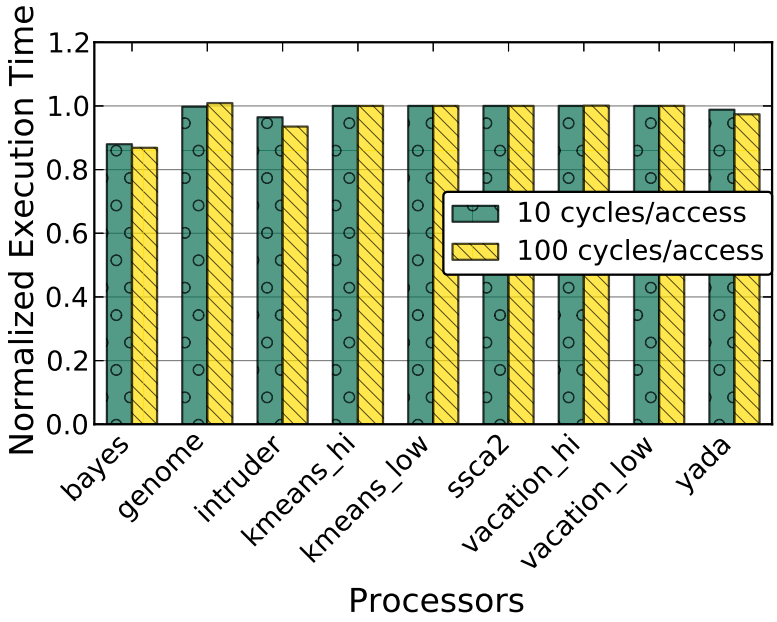


Figure 6.12: The latency of the Overflow Buffer (OB) barely affects the EcoTM performance. The execution time is normalized to the configuration with a 0 cycles per overflow buffer access. We show the results for the 32-core configuration.

6.5 Conclusions

In this chapter, we present an extension of our previous work in EazyHTM. We present EcoTM, that provides support for unbounded transactions and reduces the conflict-detection traffic on interconnects. EcoTM provides precise conflict detection, while using a minimal amount of conflict-detection metadata. EcoTM reduces the amount of metadata by classifying cache lines to the common non-conflicting and the very uncommon conflicting. The non-conflicting cache lines need only the minimal amount of metadata. EcoTM does the classification automatically, without requiring any annotations from the programmer, and dynamically, during program execution.

EcoTM's base hardware mechanisms support all current conflict management strategies: eager, lazy, and eager-lazy. This gives EcoTM both a performance and a cost-effectiveness advantage over the alternative unbounded-HTM proposals. Our evaluation indicates that EcoTM needs less metadata, and provides significantly better performance than the state-of-the-art unbounded HTMs.

Since EcoTM features precise (instead of approximative) conflict detection,

EcoTM does not suffer from false conflicts as many other unbounded HTMs do. Typical unbounded HTMs detect conflicts using Bloom-filter signatures, which sometimes report false conflicts, and this may result in aborting non-conflicting transactions. The probability of false conflicts in Bloom-filter rapidly increases with larger transactions, which becomes particularly important if we want to support large transactions of the future workloads.

7

Related Work in Hardware Transactional Memory

The sudden shift from single-core to multi-core processors caused an intensive research in transactional memory, as well as in other synchronization mechanisms. As a result, the TM and HTM topic in particular presents many of interesting and useful techniques.

In this section, we will present the most related work with this dissertation. We summarize the previous unbounded HTM proposals in Table 7.1, and explain each of them in more details in the following text.

7.1 Related work in bounded HTMs

Moore et al. [55] propose LogTM and describe a taxonomy of TM systems based on version management and conflict detection. They place Log-TM and Unbounded TM [6] into eager HTMs. They also classify Large TM [6] and Virtual TM [62]

	Unbounded?	Conflict management	Conflict detection metadata
TCC [32]	No	**lazy**	private cache: 2 bits per line
UTM/LTM [6]	**Yes**	eager	software + acceleration
VTM [62]	**Yes**	eager	fixed size Bloom filter
PTM [24]	**Yes**	eager	software + acceleration
XTM [25]	**Yes**	**lazy**	none, compares all data values
Scalable-TCC [23]	No	**lazy**	private cache: 2 bits per line
LogTM-SE [79]	**Yes**	eager	fixed size Bloom filter
OneTM [15]	**Yes**	eager	all caches: 16 bits per line
FlexTM [69]	**Yes**	**eager-lazy**	fixed size Bloom filter
TokenTM [17]	**Yes**	eager	all caches: 16+ bits per line
LiteTM [5]	**Yes**	eager	all caches: 2 bit per line + software support
DynTM [48]	**Yes**	**eager-lazy** for core-local, eager for overflowed TXs	fixed size Bloom filter
Pi-TM [56]	No	**eager-lazy**	private cache: 3 bits per line
EazyHTM	No	**eager-lazy**	private cache: 2 bits per line
EcoTM	**Yes**	**eager-lazy**	all caches: 2 bits per line

Table 7.1: An overview of related HTM mechanisms. A desired HTM is *unbounded*, has *eager-lazy* conflict management, and has *small* conflict detection metadata (up to 2-3 bits per cache line)

as eager HTMs that do lazy version management, and TCC [32] as a purely lazy HTM.

TCC [32] was the first hardware transactional memory with lazy conflict detection and lazy conflict resolution. However, it incurs two bottlenecks. First, TCC utilizes a single common bus between processors. Second, all commits in TCC are serialized with a commit token, which has to be acquired by a transaction at commit time.

Scalable TCC [23] enhances the original TCC proposal. Scalable TCC is also a lazy HTM, with both lazy conflict and version management. Scalable TCC improves upon TCC by supporting a more scalable directory protocol, and partially concurrent transaction commits. The transactions which commit to different directories may commit in parallel. Scalable TCC also introduces a new coherence protocol, as an alternative to the common MESI/MOESI cache coherence protocols. Scalable TCC assumes that execution is always transactional, and non-transactional code is converted to implicit transactions. This adds pressure to the importance of being able to perform commits in parallel. However, Scalable TCC is limited in its scalability by the number of directories, and with a small number of directories, commits may be often serialized. Typical existing chip-multiprocessor implementations have one, or a few directories. Unlike Scalable TCC, EazyHTM is designed to work as an extension to a traditional directory protocol. EazyHTM allows truly-parallel commits, rather than being limited by the number of directories present in the system. Lastly, unlike Scalable TCC, EazyHTM has explicit transactional and non-transactional modes that do not require implicit transactions.

Shiraman et al. [69] proposed FlexTM, which was the first to provide both eager and lazy conflict management. FlexTM manages conflicts either eagerly or lazily depending on their type. Write-write conflicts are always resolved eagerly, and read-write conflicts are left to the programmer to resolve. The programmer can decide to resolve read-write conflicts lazily. FlexTM detects conflicts eagerly and resolves write-write conflicts eagerly, it also uses Bloom-filter signatures for conflict detection. In contrast, EazyHTM and EcoTM detect conflicts precisely and eagerly, and resolve all conflicts lazily, which reduces or eliminates false conflicts and provides better overall scalability.

Our intention with EazyHTM was to spur the advancements and innovation in

hardware support for TM. We can see that the protocol presented in EazyHTM was generally well accepted. The work has been cited by many researchers since its publication, and there are also some improvements to the original protocol.

For example, Titos, Negi et al. [28, 56] recently proposed Pi-TM, that reduces the conflict-detection traffic on the interconnection network. Pi-TM adds an additional Pi (Pessimistic Invalidation) state to private caches. The authors of Pi-TM came to a similar conclusion as we did in EcoTM – that the number of conflicting lines is far smaller than the number of non-conflicting lines. For non-conflicting lines, Pi-TM has core-local transactional reads, while for conflicting lines, Pi-TM performs full conflict detection. Pi-TM authors evaluated that the change reduces the number of messages on the network by about 20% compared to the baseline EazyHTM. Similarly to the optimizations we present for EazyHTM, ZEBRA **??** identifies the contended lines and handles them differently. ZEBRA puts contended lines to a special buffer, while the non-contending lines are stored in shared cache. The old values for the non-contended lines are maintained on the side. We expect that future brings us more improvements to the current EazyHTM protocol, and that Eager-Lazy HTMs become the standard conflict-management approach.

7.2 Related work in unbounded HTMs

While the area of best-effort HTMs is well studied, the unbounded HTMs still have to catch up in some segments. Almost all of unbounded HTMs use eager conflict resolution for overflowed transactions, which restricts the scalability. Some proposals approach Bloom-filter signatures for conflict detection, but this sometimes give false conflicts, resulting in the aborts of non-conflicting transactions. The probability of false conflicts in Bloom-filter rapidly increases with larger transactions.

This makes the usage of Bloom-filters undesirable for the tentative future transactional workloads, which can have much larger transaction from the ones in current synthetic TM benchmarks.

UTM [6] is one of the first unbounded HTM proposals. UTM stores the conflict-detection metadata in software and accesses it using hardware extensions. Although it is an eager HTM, it keeps the speculative data private, as lazy HTMs do. It focuses on capabilities like closed transaction nesting and context switches at the

expense of performance and complexity.

VTM [62] is the first unbounded HTM with eager conflict resolution and lazy version management. VTM proposes using a counting Bloom filter (XADT filter) for conflict detection. Counting Bloom filter is known to be complex and expensive to implement in hardware, in addition to creating false conflicts. Similarly to UTM, VTM assumes that overflowed transactions are rare and focuses on correct execution rather than high performance, as EcoTM does.

PTM [24] is an eager HTM that stores the overflown data in dedicated physical pages called Shadow Page Tables (SPTs). PTM tracks all transactional metadata in *software-managed* double-linked lists (TAVs) that have to be iterated for every conflict detection with overflown transactions. As iterating SPTs and TAVs entails potentially slow memory access (and potentially multiple line evictions), a dedicated hardware accelerator is proposed to buffer SPT and TAV search results. The hardware accelerator includes a 512 entry and 2048 entry CAMs for SPT and TAV cache, respectively. In contrast with EcoTM, PTM's hardware accelerators are used for all transactional cache lines and are much more likely to be overflowed.

XTM [25] avoids most of the transactional metadata by detecting conflicts using data comparison. When overflowed transactions want to commit, XTM compares the contents of all the pages touched during transactional execution with the current page contents. If the contents are the same, the commit is allowed. For large transactions, this imposes a significant execution time overhead. In contrast, EcoTM detects all conflicts during transaction execution and therefore its conflict detection becomes a much simpler bit-check.

LogTM-SE [79] is an eager HTM that detects conflicts using Bloom-filter signatures. The main motivation of LogTM-SE is separating the transactional metadata from caches. This simplifies the support for context switching, migration and paging. However, the use of Bloom-filters hurts the performance with large transactions, while the eager conflict resolution does not perform well under medium and high contention workloads.

OneTM [15] separates the conflict detection metadata into a separate hardware structure in order to support having transactions larger from caches. However, OneTM is an eager HTM, and furthermore it allows only one transaction that overflows the private cache to execute at a time.

FlexTM [69] provides Bloom-filter-based hardware support for STMs. The Bloom-filters are used for conflict detection, and this decision significantly affects the execution time, as we show in Section 6.4. While this can simplify and accelerate STMs, using Bloom filters for conflict detection is far from ideal, as we show in this paper.

TokenTM [17] is an eager HTM that eliminates the false conflicts present in LogTM-SE by adding a significant amount of metadata to each cache block. While TokenTM needs at least 16 bits per cache block, EcoTM needs only 2 bits per cache block. The performance of TokenTM is close to the eager-perfect HTM, which we compare to EcoTM in Section 6.4.

LiteTM [5] improves on TokenTM by reducing the large amount of metadata required by TokenTM, and uses software functions to infer related information. It is on average 4% slower from TokenTM (and eager-perfect HTM), and 10% in the worst case. In contrast, EcoTM does not require any software support and has better performance than eager-perfect HTM.

DynTM [48] adds to the Bloom-filter signature only the lines evicted from the L1 cache. This reduces the number of entries in the Bloom-filter signatures, and therefore reduces the probability of false conflicts by the signatures. The core-local transactions can dynamically switch between eager or lazy conflict management, but overflowed transactions are forced to eager conflict management.

A lot of recent work focuses on optimizing the Bloom-filter signatures. For example, Quislant et al. [61] propose using location-sensitive hash functions that map the nearby memory locations to the same bits of a signature. Yen et al. [80] provide more efficient hash functions for the signatures, and allow a programmer to define locations that might create conflicts, and insert only these locations into the signature. In contrast with these proposals, EcoTM provides an automatic mechanism that does not need any effort from a programmer, and that will perform well with any future TM workload, even for extremely large transactions, for as long as the transactions have few real conflicts.

8

Conclusions and the Future of TM

This dissertation demonstrates how small modifications of cache coherence protocols on chip multiprocessors allows us to implement a well performing and efficient unbounded HTM system.

We proposed a novel HTM system, *EazyHTM*, which detects conflicts eagerly and resolves them lazily. EazyHTM makes a good trade-off between hardware complexity, the HTM performance, and its capabilities. After applying several optimizations to the initial EazyHTM design, we obtained a significant reduction in the total number of conflict detection messages by ignoring those for read-only cache lines. The EazyHTM protocol provides a complete and exact snapshot of all conflicts during transaction execution. Having this snapshot presents a wealth of useful information which could be leveraged for further research into transaction prioritization, performance optimizations and power management.

We further improve EazyHTM by including support for unbounded transactions, and by reducing the conflict-detection traffic on interconnects. We name this improved implementation *EcoTM*. EcoTM provides precise conflict detection, while

using a minimal amount of conflict-detection metadata. EcoTM achieves this by distinguishing the uncommon conflicting from the common non-conflicting cache lines automatically, without requiring any annotations from the programmer, and dynamically, during program execution. EcoTM's base hardware mechanisms support all current conflict management strategies: eager, lazy, and eager-lazy. This gives EcoTM both a performance and a cost-effectiveness advantage over the alternative unbounded-HTM proposals. We evaluate EcoTM and conclude that EcoTM needs less metadata, and provides significantly better performance than the state-of-the-art unbounded HTMs.

8.1 The future of TM

TM initially made a lot of hype in the community, by promising to bring parallel programming to the engineering masses. In theory, TM provides a simple way to write well-performing and correct multi-threaded programs, optimal for execution on multi-core processors. A software developer creates some threads, add "atomic" constructs in some places, and the TM system ensures the correctness and the performance of a multi-threaded program.

Recent research pointed out several problems with TM-based parallel programming. Most of all, TM (as it is today) does not eliminate many important problems associated with parallel programming. Since existing TM implementations are in software, the overheads of the implementations take the performance of TM far below the performance of locks. While hardware TM implementations provide good performance, they either depend on complex hardware logic or provide limited functionality. For example, many HTMs do not efficiently support large transactions, input/output (IO), interrupts, or context switches.

The adoption of transactional memory was always limited by poor performance of software implementations, and by complexity of hardware support. This led some researchers to start giving up on TM. I believe that TM will be used in the future. Maybe not *in a way, or where* it was initially envisioned, but it will very likely be used in some way.

The research on transactional memory appears to have achieved something. IBM recently announced [18] that the processors in their new supercomputer Blue-

Gene/Q will include hardware support for transactional memory. BlueGene/Q will power the 20 petaflops Sequoia supercomputer in Lawrence Livermore National Labs. This will be the ultimate test for transactional memory. Although the preliminary evaluations are highly positive, practical applications will show if TM is a versatile solution to many of the issues that currently make highly scalable parallel programming a difficult task.

BlueGene/Q is a 64-bit PowerPC-based system-on-chip that has 18 processor cores, and each core is a 4-way multi-threaded PowerPC A2 design. The processor chips have 1.47 billion transistors. Sixteen processor cores will be used for running actual computations, one will be used for running the operating system, and one core is spare and will be used to improve the reliability of the chip. The processors will run at 1.6GHz, making each chip capable of a executing 204.8 GFLOPS within a 55 W power envelope. The processor chips also include memory controllers and I/O connectivity.

BlueGene/Q is the first commercial processor to include hardware support for transactional memory, although Sun's Rock processor was supposed to do the same but was canceled when the company was purchased by Oracle. The HTM in BlueGene/Q has little or no performance penalty, meaning that we will soon see if TM is useful in practice as it is in theory. The HTM hardware can alternatively be used for speculative execution, providing IBM with a fallback functionality for the hardware.

As we will show here, hardware support for TM might be reused in many other places. By analyzing the problems faced by modern computer science, we can reason about such possible applications of the TM hardware.

8.1.1 The perspective of hardware developers

Finding the optimal synchronization constructs for multi-threaded applications is not easy, and will require strong collaboration of software and hardware industry. Software developers are always on a lookout for better synchronization constructs, being constantly faced with difficult programming and/or with poor performance of parallel programs. On the other side, hardware developers face a lack of focus and dedication of software developers, and therefore hesitate to provide stronger

hardware support for a particular synchronization construct, including TM.

For any hardware extension (including TM), hardware developers are constrained by the following requirements:

- **Optimally dedicate hardware resources (both transistors and area)** – give more resources to the frequently used functionality, give less resources to the rarely used functionalities
- **Avoid supporting legacy ISA** – avoid adding instructions that will be used rarely in future applications (but will still have to be supported for compatibility)
- **Minimize power consumption** – and to keep the overall power consumption below the power envelope
- **Minimize verification effort** – avoid adding hardware that is too complex to verify

To satisfy these requirements, a TM acceleration hardware should:

1. occupy small chip area,
2. be fairly simple to verify and easy to implement in hardware,
3. be flexible enough to provide advantages to both for current and future software,
4. have good performance with few threads, that is, have low overhead over non-TM execution, and
5. have good performance with many threads, that is, have good scalability.

Since it is difficult to satisfy all the requirements, it would be ideal if the TM acceleration could be used for general (non-TM) purposes as well. The possibility of improving the performance, simplicity, or reliability of a wider range of computer applications would make the processor manufacturers more determined in including this hardware support in future processors.

Some previous proposals for hardware acceleration for TM satisfy many of the requirements. For example, Saha et al. [64] proposed HASTM, which associates tags with the lines in private caches. The tags associated with a line are deleted if the line is evicted from the private cache. If no tagged line is evicted during transaction execution, the transaction does not have to validate, and thus can avoid this time-consuming procedure. As another example, Harris et al. [35] proposed dynamic multi-purpose hardware filter, which can reduce redundant work in: STMs, garbage collectors, memory protection mechanisms, and possibly other software mechanisms. For example, a line logged once by an STM does not have to be logged again in the same transaction. Unfortunately, these TM accelerations fail to deliver sufficiently good performance, and this drives the research on pure-hardware TM.

One way to classify the possible future usages of TM hardware could be based on the parallelism. The TM hardware could be used for: (1) sequential code, which we explore in Section 8.1.3, (2) converting the sequential code to parallel, explored in Section 8.1.4, or (3) optimizing the parallel code, explored in Section 8.1.5.

8.1.2 The interface to the TM hardware

TM hardware is typically hidden from software, providing only a high-level interface for beginning, committing, or aborting a transaction. Exposing a lower-level interface to the TM hardware could create new and interesting applications of TM. However, the exposed HTM interface should be high enough to allow future modifications of the TM hardware without having to change the interface.

The following functionalities could be exposed to software:

- Conflict detection,
- Conflict resolution,
- Writes (that is, creating the write set),
- Reads (that is, creating the read set).

The following text tries to present the motivations for exposing certain functionalities of TM.

8.1.3 New uses of HTM in sequential code

In this section, we present several examples where sequential applications could benefit from certain HTM functionalities.

Garbage collection

Garbage collection (GC) simplifies the life of a programmer by eliminating a common source of bugs in programs, where a program keeps allocating, without releasing unused memory. GC automatically discovers and releases (frees) unused variables and objects. The downside of the GC is that the discovery of unused variables is a difficult task, and may significantly increase the execution time of an application.

There have been many hardware proposals for hardware accelerations of GC mechanisms. Many of these proposals are based on tagged memory [44, 53, 54, 76]. In tagged memory, each memory block (for example, cache line) has an associated tag. A tag is a special number assigned to a particular memory block, and the number/tag value is determined by a specific GC implementation.

Generational GCs are one type of GCs that exploits the empirical observation that the most recently allocated objects are more likely to become unreachable first. A generational GC separates the memory locations into an "old" and "young" generation. If a variable from the old generation references a variable from the young generation, the young variable becomes a candidate for promoting into the old generation.

The old-to-young references in generational GCs are detected using write barriers, which can be replaced with conflict detection hardware of an HTM. The detection initializes by adding an old generation to the read set of a transaction. After that, the HTM detects writes to the old generation as conflicts. For this, an HTM should provide the conflicting addresses to the software, and to allow software to resolve the detected conflicts.

The approach does not use the version management or the conflict resolution support of an HTM. In fact, automatic conflict resolution is counter-productive, since a detected write to the old generation should be ignored after GC checks it. Ignoring the conflict allows the GC to continue executing without having to

re-initialize.

Memory protection

Memory protection is motivated by increased concern on the privacy, which made it an important research area. Memory protection can be applied to: (1) memory reads, (2) memory writes, (3) control flow, or (4) a combination of the previous. Memory protection typically restricts a code segment to a predetermined "safe" memory area. Memory protection aims to improve the stability (providing better resistance to software bugs), and also the security (preventing malicious software attacks).

There are various methods for providing memory protection. For example, CFI by Abadi et al. [1] checks that a program's control flow graph is consistent with a statically-computed safe control flow graph. XFI by [27] extends CFI with checks on data accesses. DFI by Castro et al. [22] and WIT by Akritidis et al. [4] propose instrumenting the source code with dynamic checks to verify that data accesses are in accordance with the static analysis of a program's correct behavior.

There are many ways to accelerate memory protection mechanisms. One way would be by hardware checking of memory accesses against the (predetermined or runtime generated) "safe" addresses. Another way could be by eliminating or reducing the repeating checks, that is, the checks for re-accessing the addresses. Since read and write sets of a transaction ideally contain unique addresses, they can provide the addresses accessed by a block of code. The software memory protection mechanism can analyze these accesses, and if all memory accesses are clean, it can commit the transaction. In case of an illegal memory access, it can abort the transaction and raise an exception.

A possible TM hardware could allow the software to inspect and modify the read and write set of a transaction. A transaction validation could be a dedicated hardware functionality, for example, detect when a transaction accesses addresses addresses outside of the software-configured write set. For additional flexibility, it would be good to support transaction validation in software.

Reliability

Hardware reliability becomes increasingly important with new chip manufacturing technologies. Beside providing higher efficiency, smaller transistors and lower operating voltages result in higher probability of errors.

Errors can be one-time (transient), or permanent. One-time errors can occur if, for example, a cosmic or ambient radiation excites a single transistor. Since the transistor size decreases with new technological processes, they also have higher probability of exciting a transistor, that is, changing its state from off to on. Permanent errors occur if a certain part of the processor hardware fails.

TM can detect both one-time and permanent errors. A simple approach can be to re-execute the same code, with the same input, on different processor cores, and to compare the output. The outputs can be compared entirely in software, or with help of some other hardware support. For example, additional hardware support could calculate hashes from the outputs of different executions, and then the software can compare only hashes instead of the complete outputs.

A possible TM hardware could allow the software to inspect the read and write set of a transaction. A transaction validation could be a dedicated hardware functionality, for example, detect two transactions have different write sets. However, for additional flexibility, it would be good to support transaction validation in software.

Increasing the size of basic blocks

Basic blocks are sequences of instructions that do not contain any jump (branch) instructions, and that are not a target of jump instructions. As an exception, the first instruction of a basic block may be a target of a jump, or the last instruction may be a jump instruction. Every instruction in a basic block always executes before all those in later positions.

Bigger basic blocks are desirable for many reasons. For example, they can reduce the pressure on hardware branch predictors, and they allow better overall compiler optimizations. The size of some basic blocks could be increased by joining the neighbouring basic blocks. The joining can be done if we can guarantee that a jump between the neighbouring basic blocks will always have the same outcome.

This is obvious for unconditional jumps, however, some conditional jumps can also be optimized out. The conditional jumps will have the same outcome as long as the conditional value is unchanged.

We can employ TM in at least two ways. First way is to speculatively (in a transaction) execute the code after the conditional jump, assuming that the jump will have a certain outcome. If the prediction was correct, the transaction is committed. In case of a mis-prediction, the outcome of the jump is recalculated, and the transaction is aborted and re-executed. The whole mechanism is similar to the speculative execution of code in modern out-of-order processors, only the TM can have much larger transactions and therefore the ouf-of-order logic can be simpler.

Another way to employ TM is to use it for monitoring the conditional variables. A conditional variable determines the outcome of (one or more) conditional jumps, and the conditional jumps will have the same outcome until the value of the conditional variable changes.

The conflict detection in HTM hardware could be used to efficiently monitor the changes in conditional variables. All conditional variables can be added to the read set of a transaction. If an HTM detects a change of a conditional variable, it "alerts" the dynamic recompiler, which then re-evaluates the affected conditional jump(s). The transaction that holds the conditional variables can continue executing after the code had been recompiled. By relying on (H)TM instead of instrumenting all memory writes, we lower the overhead of extending the basic blocks.

A possible TM hardware could allow the software (the dynamic recompiler) to inspect the read and write set of a transaction. The software should get notifications when a transaction gets a conflict, and it should be possible to resolve conflicts in software.

8.1.4 New uses of HTM: migrating from sequential to parallel code

One way to make a parallel application is by basing it on the sequential implementation of the same functionality. The approach consists of two steps:

1. Profiling the execution, that is, finding the computation-intensive code segments, and

2. Analyzing data dependencies between code segments.

Execution profiling can be efficiently done with existing tools, since some profiling tools statistically analyze the execution, and this increases the execution time only minimally. One such profiling tool is GNU gprof [29].

Analyzing of data dependencies between code segments is a more difficult problem. A parallel execution of code segments is possible only if they have a small amount of data dependencies. At the moment, a developer needs to supply supply the information on the data dependencies of each code segment to the runtime. This requires that a developer becomes familiar with the algorithms and the architecture of the application. To counter this, a lot of recent work tries to analyze the program execution and to find the data dependencies automatically. Current techniques use binary instrumentation, or processor emulators, which can significantly increase the execution time of an application.

A possible TM hardware could allow the software to inspect the read and write set of a transaction. By allowing the software to inspect the read and write set, analyzing the data dependencies can be done almost without any effort or overheads.

8.1.5 New uses of HTM in parallel code

Data race detection

Due to data races, successive reads of the same variable may (unexpectedly) return different values. This can lead to bugs that are very difficult to discover in a parallel multi-threaded program. A more complex case of a data race may also exist in an object, or in a group of objects. In this case, an object or a group of objects can contain an inconsistent state at some point of execution. The consistent and inconsistent states are entirely defined by software.

Data races can occur only when a variable is used by more than one thread, while at least one of the threads writes to it. Coincidentally, this execution scenario is identical to a conflict in TM terminology. This means that we could use the TM conflict detection, for detecting data races. Since the execution of entire thread (parallel section) would need to be executed in a transaction, the TM needs to support very large transaction.

A possible TM hardware should support very large transactions. It should also provide the software with the conflicting addresses. Since there are no active transactions, the conflicting addresses are, at the same time, the addresses that have potential data races.

Optimizing parallel applications

A parallelized application may have much worse performance compared to the initial sequential application. The performance overheads specific to parallel applications are:

- Code synchronization (for example: contention on critical sections, barriers)
- Implicit data synchronization (for example, cache coherence)
- Explicit data synchronization (for example, message passing)

In case these overheads exceed the advantages of parallel execution, a parallel application will likely execute longer than the sequential version of the same application.

Quantifying the overheads often requires ad-hoc methods, which in general do not give exact results. For example, a developer may add a counter before trying to take a lock, in order to count how many times a lock was taken.

Recent processors include hardware counters that can be used by software developers to minimize cache misses, TLB misses, branch miss-prediction, etc. Some recent compilers with dynamic recompilation even use these hardware counters to re-optimize the code while the program is executing.

However, hardware support for optimizing parallel execution is still in its early stages. It is clear that, to improve the efficiency of parallel applications, some kind of hardware support is absolutely necessary. Adding hardware support for profiling parallel applications would be a great help in this. Examples of such support include:

- Find variables or cache lines that exhibit a ping-pong effect (frequently bounce between processor-local caches)

- Find variables or cache lines evicted due to the coherence requests (potential data races)
- Find code locations that create conflicts between transactions
- Hardware counters for the number of transactions started, committed, and aborted (split to causes: insufficient hardware resources, conflicts, and interrupts)

Task-based programming

Task-based programming [10, 60] is becoming increasingly popular. In task-based programming, developer splits a program into tasks. Each task defines the locations it requires (the input) and the locations it produces (the output). Based on the input and output, the runtime determines an order in which the tasks can be executes, and determines the tasks that can execute in parallel.

In contrast with regular parallel programs, the results of task-based programs are deterministic. A task-based parallel program can be created by annotating a sequential application, and the execution of the initial sequential and the resulting task-based parallel application has to be identical.

However, the inputs and the outputs of the tasks have to be defined in advance, for the runtime to be able to schedule the tasks. A failure to correctly define the inputs or the outputs can lead to non-deterministic or incorrect execution. Unfortunately, this is the main source of problems with task-based programming, since some inputs or outputs of a task may be hidden, or difficult to understand from the source code.

TM hardware could simplify a development of task-based programs, by helping the discovery of data dependencies between tasks. A developer provides all known task dependencies to the runtime, and the runtime can schedule tasks using provided dependencies. Each task can be executed in a transaction, and the TM hardware tracks all memory accesses of the task. When the task completes, the TM hardware should allow the developer to inspect the read and write set of a transaction. This information allows a developer to improve the task dependencies, and to repeat the process until he gets a complete list of dependencies.

9

Publications

Saša Tomić, Ege Akpinar, Tim Harris, Adrián Cristal, Osman Unsal, Mateo Valero.
EcoTM: Economical Conflict-Driven Unbounded Hardware Transactional Memory. *Under submission*

Srđan Stipić, **Saša Tomić**, Ferad Zyulkyarov, Adrián Cristal, Osman Unsal, Mateo Valero.
TagTM - Accelerating STMs with hardware tags for fast meta-data access . *Proceedings of the Design, Automation and Test in Europe Conference (DATE), March 2012*

Saša Tomić, *Adrián Cristal, Osman Unsal, Mateo Valero.*
Rapid Development of Error-Free Architectural Simulators using Dynamic Runtime Testing. 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), *October 2011*

Ege Akpinar, **Saša Tomić***, Adrián Cristal, Osman Unsal, Mateo Valero.*
A Comprehensive Study of Conflict Resolution Policies in Hardware Transactional Memory. 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, June 2011*

Y. Afek, U. Drepper, P. Felber, C. Fetzer, V. Gramoli, M. Hohmuth, E. Riviere, P. Stenstrom, O. Unsal, W.M. Moreira, D. Harmanci, P. Marlier, S. Diestelhorst, M. Pohlack, A. Cristal, I. Hur, A. Dragojevic, R. Guerraoui, M. Kapalka, **S. Tomić***, G. Korland, N. Shavit, M. Nowack, and T. Riegel.*
The Velox Transactional Memory Stack. IEEE Micro 30*, September 2010*

Tim Harris, **Saša Tomić***, Adrián Cristal, Osman Unsal.*
Dynamic Filtering: Multi-Purpose Architecture Support for Language Runtime Systems. 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2010*

Saša Tomić*, Cristian Perfumo, Chinmay Kulkarni, Adria Armejach, Adrián Cristal, Osman Unsal, Tim Harris, Mateo Valero.*
EazyHTM, Eager-Lazy Hardware Transactional Memory. 42nd International Symposium on Microarchitecture (MICRO)*, Dec 2009*

Saša Tomić*, Adrián Cristal, Osman Unsal, Mateo Valero.*
Hardware Transactional Memory with Operating System Support, HTMOS. Workshop on Highly Parallel Processing on a Chip in conjunction with Euro-Par (HPPC)*, Aug 2007*

References

[1] *Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In* Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM. ISBN 1-59593-226-7. DOI: 10.1145/1102120.1102165. Cited on page: 127*

[2] *Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In* PPoPP '09: Proc. 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 185–196, February 2009. DOI: 10.1145/1504176.1504203. PDF:* `http://research.microsoft.com/en-us/um/people/tharris/papers/2009-ppopp.pdf`*. Cited on page: 23*

[3] *Alain Abran, James W Moore, Robert Dupuis, RL Dupuis, and L L Tripp. Guide to the software engineering body of knowledge (swebok).* 2004 ed P Bourque R Dupuis A Abran and JW Moore Eds IEEE Press*, page 204, 2001. Cited on page: 50*

[4] *Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In* Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3168-7. DOI: 10.1109/SP.2008.30. Cited on page: 127*

[5] *Syed Ali Raza Jafri, Mithuna Thottethodi, and T. N. Vijaykumar. LiteTM: Reduc-*

ing transactional state overhead. In Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA-17)*, January 2010. PDF:* http://cobweb.ecn.purdue.edu/~vijay/papers/2010/litetm.pdf*. Cited on page: 116, 120*

[6] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In International Symposium on High-Performance Computer Architecture*, pages 316–327, February 2005. DOI: 10.1109/HPCA.2005.41. PDF:* http://www.cag.csail.mit.edu/scale/papers/utm-hpca2005.pdf*. Cited on page: 115, 116, 118*

[7] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. Commun. ACM*, 52:56–67, October 2009. ISSN 0001-0782. DOI: 10.1145/1562764.1562783. Cited on page: 1*

[8] David Becker, Raj K. Singh, and Stephen G. Tell. Readings in hardware/software co-design. pages 550–555, 2002. Cited on page: 58

[9] F. Bellard. Qemu, a fast and portable dynamic translator. In Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005. Cited on page: 36, 37*

[10] Saniya Ben Hassen, Henri E. Bal, and Ceriel J. H. Jacobs. A task- and data-parallel programming language based on shared objects. ACM Trans. Program. Lang. Syst.*, 20:1131–1170, November 1998. ISSN 0164-0925. DOI: 10.1145/295656.295658. Cited on page: 132*

[11] Nathan Binkert, Ronald Dreslinski, Lisa Hsu, Kevin Lim, Ali Saidi, and Steven Reinhardt. The M5 simulator: Modeling networked systems. IEEE Micro*, 26 (4):52–60, 2006. DOI: 10.1109/MM.2006.82. Cited on page: 25, 54, 105*

[12] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM*, 13(7):422–426, 1970. ISSN 0001-0782. DOI: 10.1145/362686.362692. Cited on page: 18*

[13] *Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In* Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2005. PDF:* http://www.cis.upenn.edu/acg/papers/wddd05_atomic_semantics.pdf. *Cited on page: 23*

[14] *Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Subtleties of transactional memory atomicity semantics.* Computer Architecture Letters*, 5(2), November 2006. DOI: 10.1109/L-CA.2006.18. PDF:* http://www.cis.upenn.edu/acg/papers/cal06_atomic_semantics.pdf. *Cited on page: 68*

[15] *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* Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007. DOI: 10.1145/1273440.1250667. Cited on page: 17, 18, 19, 20, 116, 119*

[16] *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* Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007. DOI: 10.1145/1250662.1250674. PDF:* http://www.cs.wisc.edu/multifacet/papers/isca07_pathologies.pdf. *Cited on page: 7, 10, 21, 22, 79*

[17] *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* Proceedings of the 35th Annual International Symposium on Computer Architecture*, June 2008. DOI: 10.1109/ISCA.2008.24. PDF:* http://www.cs.wisc.edu/multifacet/papers/isca08_tokentm.pdf. *Cited on page: 17, 18, 19, 20, 105, 116, 120*

[18] *Luigi Brocher. Innovative technologies for power management based on power architecture, 2011. PDF:* https://www.power.org/events/

PowerWebinar-03-29-11/IBM_March_29_Webinar_-_Dr._Luigi.pdf. *Cited on page: 122*

[19] 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 Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007. DOI: 10.1145/1250662.1250673. PDF:* http://tcc.stanford.edu/publications/tcc_isca2007.pdf*. Cited on page: 18, 19, 74*

[20] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008. DOI: 10.1109/IISWC.2008.4636089. PDF:* http://tcc.stanford.edu/publications/tcc_iiswc2008.pdf*. Cited on page: 27, 55, 74, 78, 105, 109*

[21] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? Communications of the ACM*, 51(11): 40–46, November 2008. DOI: 10.1145/1400214.1400228. PDF:* http://www.cis.upenn.edu/~blundell/stm-cacm2008.pdf*. Cited on page: 6*

[22] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 11–11, Berkeley, CA, USA, 2006. USENIX Association. Cited on page: 127*

[23] 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 HPCA*, pages 97–108, 2007. DOI: 10.1109/HPCA.2007.346189. Cited on page: 66, 79, 116, 117*

[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* ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 347–358. ACM, 2006. DOI: 10.1145/1168857.1168901. Cited on page: 17, 18, 116, 119*

[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* ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM Press, October 2006. DOI: 10.1145/1168857.1168903. PDF:* `http://tcc.stanford.edu/publications/tcc_asplos2006.pdf`*. Cited on page: 18, 19, 116, 119*

[26] *Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why stm can be more than a research toy.* Commun. ACM*, 54:70–77, April 2011. ISSN 0001-0782. DOI: 10.1145/1924421.1924440. Cited on page: 6*

[27] *Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: software guards for system address spaces. In* Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. Cited on page: 127*

[28] *José Rubén Titos Gil.* Hardware Techniques for High-Performance Transactional Memory in Many-Core Chip Multiprocessors*. PhD thesis, Universidad de Murcia, 2011. Cited on page: 118*

[29] *S.L. Graham, P.B. Kessler, and M.K. Mckusick. Gprof: A call graph execution profiler. In* Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126. ACM, 1982. Cited on page: 130*

[30] *Lance Hammond, Brian D. Carlstrom, Vicky Wong, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency:*

Simplifying parallel hardware and software. IEEE Micro*, 24(6), Nov-Dec 2004. DOI: 10.1109/MM.2004.91. PDF:* http://tcc.stanford.edu/publications/tcc_micro2004.pdf*. Cited on page: 18, 19, 45, 56*

[31] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (TCC). In ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13. ACM Press, October 2004. DOI: 10.1145/1024393.1024395. PDF:* http://tcc.stanford.edu/publications/tcc_asplos2004.pdf*. Cited on page: 18*

[32] 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 Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, June 2004. DOI: 10.1145/1028176.1006711. PDF:* http://tcc.stanford.edu/publications/tcc_isca2004.pdf*. Cited on page: 66, 116, 117*

[33] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. Communications of the ACM*, 51(8):91–100, August 2008. DOI: 10.1145/1378704.1378725. An earlier version appeared at PPoPP '06. Cited on page: 6*

[34] Tim Harris, James Larus, and Ravi Rajwar. Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010. ISBN 1608452352, 9781608452354. Cited on page: 3, 44*

[35] Tim Harris, Saša Tomić, Adrián Cristal, and Osman Unsal. Dynamic filtering: multi-purpose architecture support for language runtime systems. In Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 39–52, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-839-1. DOI: 10.1145/1736020.1736027. Cited on page: 125*

[36] *Mary Jean Harrold. Testing: a roadmap. In* Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 61–72, New York, NY, USA, 2000. ACM. ISBN 1-58113-253-0. DOI: 10.1145/336512.336532. Cited on page: 36*

[37] *J. Held. From a few cores to many: A tera-scale computing research overview.* Intel White Paper*, 2006. Cited on page: 1*

[38] *M. Herlihy and N. Shavit. On the nature of progress.* Unpublished work*, 2008. PDF:* http://www.cs.tau.ac.il/~shanir/progress.pdf*. Cited on page: 4*

[39] *Maurice Herlihy and Nir Shavit.* The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914. Cited on page: 4, 12*

[40] *C.A.R. Hoare. An axiomatic basis for computer programming.* Communications of the ACM*, 12(10):576–580, 1969. Cited on page: 57*

[41] *Corp. Intel. Intel IA-64 Architecture Software Developer's Manual.* Itanium Processor Microarchitecture Reference for Software Optimization*, August 2000. Cited on page: 99*

[42] *D.N. Jayasimha, Bilal Zafar, and Yatin Hoskote. On-die interconnection networks: Why they are different and how to compare them. In* Technical Report at http://blogs.intel.com/research/terascale/ODI_why-different.pdf, Microprocessor Technology Lab, Corporate Technology Group, Intel Corp.*, 2006. Cited on page: 79*

[43] *R. Jindal and K. Jain. Verification of transaction-level systemc models using rtl testbenches. In* Formal Methods and Models for Co-Design, 2003. MEMOCODE'03. Proceedings. First ACM and IEEE International Conference on*, pages 199–203. IEEE, 2003. Cited on page: 36*

[44] *Robert H. Halstead Jr. and Tetsuya Fujita. Masa: A multithreaded processor architecture for parallel symbolic computing. In* ISCA*, pages 443–451, 1988. Cited on page: 126*

[45] *C. Kaner, J. Bach, and B. Pettichord.* Lessons learned in software testing: a context-driven approach. *Wiley, 2002. Cited on page: 56*

[46] *Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In* Proceedings of the 1st international symposium on Memory management*, ISMM '98, pages 176–185, New York, NY, USA, 1998. ACM. ISBN 1-58113-114-3. DOI: 10.1145/286860.286880. Cited on page: 5*

[47] *Jian Li and José F. Martínez. Power-performance implications of thread-level parallelism on chip multiprocessors. In* International Symposium on Performance Analysis of Systems and Software*, pages 124–134, 2005. DOI: 10.1109/ISPASS.2005.1430567. Cited on page: 1*

[48] *Marc Lupon, Grigorios Magklis, and Antonio González. A dynamically adaptable hardware transactional memory. In* MICRO 43: Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010. Cited on page: 90, 116, 120*

[49] *Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Scalable techniques for transparent privatization in software transactional memory. In* ICPP '08: Proc. 37th International Conference on Parallel Processing*, September 2008. DOI: 10.1109/ICPP.2008.69. PDF:* http://www.cs.rochester.edu/~vmarathe/research/papers/2008_ICPP.pdf*. Cited on page: 94*

[50] *C. May, E. Silha, R. Simpson, and H. Warren.* The PowerPC Architecture: A specification for a new family of RISC processors. *Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1994. ISBN 1558603166. Cited on page: 99*

[51] *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* Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2005. PDF:* http://tcc.stanford.edu/publications/tcc_pact2005.pdf*. Cited on page: 74*

[52] *Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In* Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 53–65, June 2006. Cited on page: 74*

[53] *Matthias Meyer. A true hardware read barrier. In* Proceedings of the 5th international symposium on Memory management*, ISMM '06, pages 3–16, New York, NY, USA, 2006. ACM. ISBN 1-59593-221-6. DOI: 10.1145/1133956.1133959. Cited on page: 126*

[54] *David A. Moon. Garbage collection in a large lisp system. In* Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 235–246, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. DOI: 10.1145/800055.802040. Cited on page: 126*

[55] *Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In* In proceedings of the HPCA-12*, pages 254–265, February 2006. PDF:* `http://www.cs.wisc.edu/multifacet/papers/hpca06_logtm.pdf`*. Cited on page: 17, 18, 19, 20, 45, 56, 102, 103, 115*

[56] *Anurag Negi, Rubén Titos-Gil, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. Pi-tm: Pessimistic invalidation for scalable lazy hardware transactional memory. In* 18th International Symposium on High Performance Computer Architecture (HPCA'2012)*. IEEE Conference Publishing Services, 2012. Cited on page: 116, 118*

[57] *Kunle Olukotun and Lance Hammond. The future of microprocessors.* Queue*, 3: 26–29, September 2005. ISSN 1542-7730. DOI: 10.1145/1095408.1095418. Cited on page: 1*

[58] *V. Pankratius, A.R. Adl-Tabatabai, and F. Otto. Does Transactional Memory Keep Its Promises?: Results from an Empirical Study. Technical Report Technical Report 2009-12, Universität Karlsruhe, Fakultät für Informatik, 2009. Cited on page: 2*

[59] *Salil Pant and Greg Byrd. A case for using value prediction to improve performance of transactional memory. In* TRANSACT '09: 4th Workshop on Transactional Computing*, February 2009. PDF:* http://transact09.cs.washington.edu/35_paper.pdf*. Cited on page: 18*

[60] *Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with StarSs.* Int. J. High Perform. Comput. Appl.*, 23:284–299, August 2009. ISSN 1094-3420. DOI: 10.1177/1094342009106195. Cited on page: 132*

[61] *Ricardo Quislant, Eladio Gutierrez, Oscar Plata, and Emilio L. Zapata. Improving signatures by locality exploitation for transactional memory. In* Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 303–312, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3771-9. DOI: 10.1109/PACT.2009.25. Cited on page: 18, 120*

[62] *Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In* Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, June 2005. PDF:* http://www.cs.wisc.edu/~isca2005/papers/08A-02.PDF*. Cited on page: 115, 116, 119*

[63] *Per Runeson. A survey of unit testing practices.* IEEE Softw.*, 23:22–29, July 2006. ISSN 0740-7459. DOI: 10.1109/MS.2006.91. Cited on page: 36*

[64] *Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In* MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196. IEEE Computer Society, 2006. DOI: 10.1109/MICRO.2006.9. Cited on page: 125*

[65] *Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In* MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*,*

pages 123–133. IEEE Computer Society, 2007. DOI: 10.1109/MICRO.2007.20. Cited on page: 18, 90

[66] Florian T. Schneider, Vijay Menon, Tatiana Shpeisman, and Ali-Reza Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In OOPSLA '08: Proc. 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 181–194, September 2008. DOI: 10.1145/1449955.1449779. PDF:* http://www.lst.inf.ethz.ch/research/publications/publications/OOPSLA_2008/OOPSLA_2008.pdf*. Cited on page: 23*

[67] Michael L. Scott, Michael F. Spear, Luke Dalessandro, and Virendra J. Marathe. Transactions and privatization in Delaunay triangulation (brief announcement). In PODC '07: Proc. 26th PODC ACM Symposium on Principles of Distributed Computing*, August 2007. DOI: 10.1145/1281100.1281160. PDF:* http://www.cs.rochester.edu/u/scott/papers/2007_PODC_mesh_BA.pdf*. Cited on page: 94*

[68] Arrvindh Shriraman and Sandhya Dwarkadas. Refereeing conflicts in hardware transactional memory. In ICS '09: Proc. 23rd international conference on Supercomputing*, pages 136–146, June 2009. DOI: 10.1145/1542275.1542299. PDF:* http://www.cs.rochester.edu/~ashriram/publications/2009_ICS_Referee.pdf*. Also available as TR 939, Department of Computer Science, University of Rochester, September 2008. Cited on page: 22, 79*

[69] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In Proceedings of the 35th Annual International Symposium on Computer Architecture*, June 2008. PDF:* http://www.cs.rochester.edu/u/ashriram/papers/2008_ISCA_FlexTM.pdf*. Cited on page: 103, 116, 117, 120*

[70] James Smith. A study of branch prediction strategies. In International Symposium on Computer Architecture*, pages 202–215, 1998. Cited on page: 22*

[71] *James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In* Proceedings of the 12th annual international symposium on Computer architecture*, ISCA '85, pages 36–44, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press. ISBN 0-8186-0634-7. DOI: 10.1145/327010.327125. Cited on page: 99*

[72] *Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory (brief announcement). In* PODC '07: Proc. 26th PODC ACM Symposium on Principles of Distributed Computing*, August 2007. DOI: 10.1145/1281100.1281161. PDF:* http://www.cs.rochester.edu/u/scott/papers/2007_PODC_privatization_BA.pdf*. Extended version available as TR-915, Computer Science Department, University of Rochester, Feb. 2007,* http://www.cs.rochester.edu/u/scott/papers/2007_TR915.pdf*. Cited on page: 94*

[73] *Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In* PPoPP '09: Proc. 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 141–150, February 2009. DOI: 10.1145/1504176.1504199. PDF:* http://www.cs.rochester.edu/u/spear/ppopp09.pdf*. Cited on page: 21*

[74] *R.M. Stallman, R.H. Pesch, S. Shebs, et al.* Debugging with GDB*. Gnu Press, 2002. Cited on page: 50*

[75] *Fuad Tabba, Andrew W. Hay, and James R. Goodman. Transactional value prediction. In* TRANSACT '09: 4th Workshop on Transactional Computing*, February 2009. PDF:* http://transact09.cs.washington.edu/4_paper.pdf*. Cited on page: 18*

[76] *David Michael Ungar.* The design and evaluation of a high performance Smalltalk system*. MIT Press, Cambridge, MA, USA, 1987. ISBN 0-262-21010-X. Cited on page: 126*

[77] *M.A. Vouk. Back-to-back testing.* Information and software technology*, 32 (1):34–45, 1990. Cited on page: 58*

[78] *D.L. Weaver and T. Germond.* The SPARC architecture manual. *Citeseer, 1994. ISBN 0130992275. Cited on page: 99*

[79] *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* HPCA '07: Proc. 13th International Symposium on High-Performance Computer Architecture*, February 2007. DOI: 10.1109/HPCA.2007.346204. PDF:* http://www.cs.wisc.edu/multifacet/papers/hpca07_logtmse.pdf*. Cited on page: 17, 18, 19, 20, 105, 116, 119*

[80] *Luke Yen, Stark C. Draper, and Mark D. Hill. Notary: Hardware techniques to enhance signatures. In* MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 234–245. IEEE Computer Society, 2008. DOI: 10.1109/MICRO.2008.4771794. Cited on page: 18, 90, 94, 120*