Hardware Approaches for Transactional Memory

MASTER THESIS

Marc Lupon Navazo
Departament d’Arquitectura de Computadors
<mlupon@ac.upc.edu>

Master CANS
Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya

June 29, 2008
Abstract

Transactional Memory is a promising Shared Memory programming model that provides non-blocking synchronization among atomic and isolated transactions. It addresses the programmability issues of lock-based applications using mechanisms that are transparent to developers. Although there are different strategies to implement these mechanisms, only environments with hardware support present better results than solutions based on fine-grain locking.

Taking a log-based HTM as a baseline, we present several HTM approaches that improve its transactional mechanisms adding a hardware buffer. Eager Version Management (EVM) and Lazy Version Management (LVM) approaches use the buffer to accelerate abort recovery, whereas Lazy Conflict Detection (LCD) delays the detection of conflicts until commit time. Hybrid Conflict Detection (HCD), a new approach that combines different conflict detection policies, has been proposed to reduce the overhead of previous approaches. A characterization of these approaches with unlimited resources has been made to determine their benefits and weakness.

Each HTM approach has been modeled with finite hardware support. Transactions that overflow hardware resources must use software transactional mechanisms, what can generate some inconsistencies among transactions. We propose a system that supports different execution modes to address these problems. We also present a technique to accelerate overflowed transactions using hardware support.

The study concludes that HTMs performance depends on the workload and the number of threads used, obtaining huge differences when contention varies. As no approach takes advantage respect to the others, an efficient HTM must support the execution of different HTM approaches.
## Contents

1 Introduction  
1.1 Objectives and Contribution ........................................... 3  
1.2 Organization .............................................................. 4

2 Transactional Memory ......................................................... 5  
2.1 Parallel Programming Models ........................................... 5  
2.2 Blocking Synchronization Problems ..................................... 6  
2.3 Transactional Memory Model ............................................. 9  
2.4 Transactional Memory environments .................................. 11  
   2.4.1 Software Transactional Memory .................................. 11  
   2.4.2 Hardware Transactional Memory .................................. 12  
   2.4.3 Hybrid Transactional Memory .................................... 13  
   2.4.4 Hardware-Accelerated Transactional Memory ................. 13

3 Hardware Transactional Memory ............................................ 15  
3.1 Access summary ............................................................ 15  
3.2 Version Management ...................................................... 17  
   3.2.1 Lazy Version Management ...................................... 18  
   3.2.2 Eager Version Management ...................................... 18  
3.3 Conflict Detection ....................................................... 20  
   3.3.1 Eager Conflict Detection ....................................... 20  
   3.3.2 Lazy Conflict Detection ....................................... 20  
3.4 Conflict Resolution ..................................................... 22  
3.5 Nested Transactions ...................................................... 23  
3.6 OS Support for HTM ...................................................... 25

4 Hardware Approaches for Transactional Memory ......................... 27  
4.1 LogTM ................................................................. 27  
4.2 Transactional Buffer ................................................... 30  
4.3 Eager Version Management ............................................ 31  
4.4 Lazy Version Management ............................................ 32  
4.5 Lazy Conflict Detection .............................................. 34  
4.6 Hybrid Conflict Detection ............................................ 38
5 Bounded Support for HTM Approaches
5.1 LogTM-SE .......................................................... 40
5.2 Bounded Eager Version Management .......................... 41
5.3 Bounded Lazy Version management .......................... 42
5.4 Bounded Hybrid Conflict Detection .......................... 44
5.5 Bounded Lazy Conflict Detection ............................ 44

6 Evaluation
6.1 Base System ......................................................... 47
6.2 Simulation Methods ................................................ 47
6.3 Benchmarks and Workloads ..................................... 49
6.4 Experimentation ..................................................... 50
   6.4.1 LogTM characterization .................................... 51
   6.4.2 Unbounded HTM approaches .............................. 56
   6.4.3 HTM approaches with bounded support ................. 63

7 Conclusions ......................................................... 68

A A transactional coherency protocol ............................ 70
## List of Figures

2.1 Lock-based program ........................................... 7  
2.2 Coarse-grain locking vs Fine-grain locking .................. 8  
2.3 Deadlock ...................................................... 8  
2.4 Transactional Memory program ................................. 9  
2.5 Transaction’s behavior ....................................... 10  
3.1 Access summary ................................................ 17  
3.2 Eager Transactional Memory ................................... 19  
3.3 Lazy Transactional Memory .................................... 21  
3.4 Lazy Conflict Resolution behavior ............................ 23  
3.5 Nested Transactions .......................................... 25  
4.1 Log process in LogTM ......................................... 29  
4.2 Eager Version Management using the Transactional Buffer . 32  
4.3 Lazy Version Management using the Transactional Buffer . 33  
4.4 Eager Conflict Detection in EVM and LVM .................. 35  
4.5 Lazy conflict detection in LCD and HCD ..................... 36  
4.6 Lazy conflict detection commit phase ....................... 37  
4.7 Hybrid Conflict Detection Behavior .......................... 39  
5.1 Hardware support unbounded EVM with logging ............... 42  
5.2 Bounded LVM behavior ....................................... 43  
5.3 Conflict with LCD and LogTM-SE ............................. 45  
5.4 Bounded LCD behavior ...................................... 46  
6.1 Simulated CMP system ......................................... 48  
6.2 Workload speed-up using different HTM approaches ......... 52  
6.3 Workload speed-up using different HTM approaches ......... 53  
6.4 Normalized time distribution using 32-cores ................. 54  
6.5 Transaction’s behavior using 32-cores ...................... 55  
6.6 Transactional time behavior using 32-cores ................. 55  
6.7 Best HTM approach speed-up ................................ 57  
6.8 Transactional cycle’s distribution using 32-cores .......... 59  
6.9 Performance of Bounded HTM approaches .................... 65  
6.10 Performance of Optimized Bounded HTM approaches ...... 67  
A.1 L1-cache transactional MESI coherency protocol ............ 71
Chapter 1

Introduction

Processors have recently found some problems that have a critical impact on their performance progression. On the one hand, the clock rate cannot be infinitely increased, as it implies higher die temperatures and more power dissipation, what create packaging problems in laptops and cooling problems in desktops or servers. Adding pipeline stages also increments the penalty produced by flow and data hazards. For these reasons, computer designers must build thermal aware processors instead of faster, deeply pipelined processors. On the other hand, Instruction Level Parallelism (ILP) seems to have reached a limit where any improvement requires a huge amount of hardware. Moreover, wire’s latency improves slower than processor’s frequency. Due to this, signals cannot be propagated in a single cycle, delaying the execution of dependent instructions [53].

Nevertheless, technology researchers are still following Moore’s Law, making smaller transistors that allow computer designers to introduce more of them on a chip. This silicon can be devoted to increase ILP, but it wouldn’t still report important benefits. Fortunately, other architecture designs are able to use these transistors in a more efficient manner.

These facts encouraged hardware developers to change their design strategy. Instead of building big processors that take up the whole chip, nowadays they build Chip Multiprocessors (CMP). A CMP consists of several processors that are placed on the same chip and share resources. CMPs can run simultaneously several sequential tasks, called threads, on different processors, exploiting Thread Level Parallelism (TLP) [20]. CMPs provide lower latency and higher bandwidth on intra-chip messages than classical Multiprocessors (single processors connected by a network, used in high performance workstations) because processors are closer. Furthermore, the clock rate can be reduced without losing throughput performance, so they are more power efficient [38].

The benefits of using a CMP have caused a revolution in the hardware industry. CMPs are becoming the norm for server, desktop and embedded markets. Niagara T2 or Intel Core Quad are some examples of CMP commercialization. This revolution has introduced a difficult question that needs to be answered in order to take advantage of the CMP configurations: how will programmers and users exploit
parallel architectures? This challenge is difficult to answer, as it involves both the hardware and the software industry.

A simple answer is to execute several independent tasks in parallel. Although this is nowadays a good solution to keep the processors busy, the execution time of an application in a CMP is the same as in a uniprocessor. This is because, even if the application is run in a CMP, it will be executed only on one processor.

Parallel programming models, which divide an application in several tasks that can be executed concurrently, have been used to develop applications for high-performance massively-parallel computers. These tasks are a subset of the application and, for that reason, the time required to execute them on a multiprocessor is shorter than the time needed when the application only uses one processor. If all the tasks are executed at the same time, the application execution time is that of the slowest task, which could be considerably shorter if the application is divided into similarly sized tasks.

Parallelizing applications seems to be the best alternative to reduce application’s execution time because it efficiently exploits CMP resources. However, current parallel programming models are not trivial and require expert programmers to develop efficient applications for parallel architectures. Thus, the key factor limiting the potential of parallel architectures is the difficulty of parallel application development. This is the main reason why, although parallel processors are commonly widespread, most software companies are still selling sequential code.

Message Passing and Shared Memory are the most-used parallel programming models. Message Passing requires explicit communication, whereas Shared Memory requires blocking synchronization to supply mutual exclusion among critical sections of different threads. Critical sections need to acquire a lock before their execution, and in case that the acquisition fails, threads wait until the lock is released [14].

Unfortunately, blocking synchronization is not the best alternative for Shared Memory programming models. Fine-grain locking applications are difficult to develop, and only expert programmers are able to implement them without race conditions. On the other hand, coarse-grain locking is easier to program, but performs poorly, because it limits parallelism. Due to this, other synchronization techniques have been researched with promising results [50].

Transactional Memory [26] is an alternative lock-free Shared Memory programming model that provides non-blocking wait-free synchronization among critical sections. A transaction is a sequence of reads and writes on shared memory data that can be safely executed in parallel with the rest of the program. The execution of a transaction is atomic, isolated, durable and consistent. When a transaction is executed, the system tracks its program state modifications to detect possible conflicts. If two concurrent transactions access the same data, one of them must be aborted. On these occasions, recovery mechanisms must be triggered to undo the modifications of the aborting transaction. At the end of a transaction, all the program state changes performed by this transaction become visible to the other transactions.
Summarizing, Transactional Memory addresses the programmability issues of blocking synchronization without sacrificing fine-grain locking performance. For that reason, Transactional Memory is a promising idea that is becoming very popular in computer architecture research. However, building a Transactional Memory environment is complex, because additional actions are required in memory accesses or in recovery mechanisms.

Different techniques can be applied to design a Transactional Memory environment. Software Transactional Memory (STM) [46] keeps transactional state by software, using atomic instructions to guarantee isolation and barriers on every memory operation to detect conflicts, which degrades performance. Hardware Transactional Memory (HTM) [22] defines new instructions and adds hardware support to guarantee transactional behavior, reducing overhead but increasing hardware complexity.

Many HTM designs provide bounded transactional support, and sometimes transactions overflow resources [13]. Recent proposals present hardware designs that allow transactions to commit even when they exceed resources [4]. Hybrid approaches [15] propose to handle these large transactions by software, whereas common-case, smaller transactions use best-effort hardware.

1.1 Objectives and Contribution

The global objective of this Master Thesis is to study how different HTM mechanisms affect the behavior of the transactional applications. Different version management, conflict detection or conflict resolution policies have been proposed in order to preserve transactional properties. Our goal is to determine the benefits and weakness of each mechanism and design a system that permits the execution of different HTM approaches, each one with its own transactional mechanisms.

This Master Thesis presents a characterization of several HTM designs. LogTM [34], a log-based HTM that upgrades the memory hierarchy with transactional modifications and stores old transactional values in a private software log, has been taken as a baseline. Although our studies show that transactional applications that use LogTM are more efficient than lock-based applications, their performance is degraded when contention is present. That is because LogTM requires a software mechanism to restore transactional state in case of abort. This fact encouraged us to experiment with alternative strategies in order to improve LogTM performance.

A system that supports different HTM approaches has been designed. All of these HTM approaches need similar hardware support, but they use different techniques to implement transactional mechanisms. Eager Version Management (EVM) accelerates LogTM aborts by using a hardware buffer to store old values. Lazy Version Management (LVM), an HTM approach that is similar than BulkTM [10] or Rock[29], stores transactional values in the hardware buffer and updates the memory hierarchy during its commit phase, but it manages contention like LogTM. Lazy Conflict Detection (LCD) models a TCC [19] environment that speculates with
transactional accesses and detects conflicts at commit time. Hybrid Conflict Detection (HCD) is a new HTM approach that mixes EVM and LCD conflict detection policies to remove some of the weak points of previous approaches.

Each HTM approach has been modeled both with unlimited and bounded support. Transactions that overflow hardware resources must be re-executed using software mechanisms, what can generate inconsistencies with transactions that use other HTM approaches. Our design solve these problems by changing the conflict resolution policy and switching non-overflowed transaction’s execution mode.

An exhaustive analysis of transactional behavior has been made to determine the benefits and the weakness of each model using transactional applications. Results show that any of our four HTM approaches performs better than LogTM, and their performance depends on the workload and the number of threads used. LCD and HCD are useful in high-contention workloads with small transactions, whereas EVM and LVM are good running applications with large transactions. As no approach takes advantage respect the others, any HTM must support multiple modes. The study also concludes that hardware support can be useful to improve the performance of overflowed transactions.

1.2 Organization

This Master Thesis is organized as follows. Chapter 2 shows the problematic with blocking synchronization and presents Transactional Memory as the best alternative to solve it. It also describes transaction’s properties and resumes main implementation philosophies. Chapter 3 summarizes HTM background, exposing how recent proposals implement transactional mechanisms. Chapter 4 models the HTM approaches, explaining how they are implemented with unbounded support. Chapter 5 presents how previous designs can overcome hardware limitations without loosing performance by exploiting best-effort techniques in common-case transactions. Chapter 6 evolutates different HTM proposals and analyzes their behavior, whereas Chapter 7 concludes.
Chapter 2

Transactional Memory

Current parallel programming models present several programming difficulties that have encouraged research on alternatives that provide better programmability without losing performance. Transactional Memory is a parallel programming model that provides an easy and intuitive paradigm based on transactions.

This chapter describes the currently dominant parallel programming models and their associated problems that have motivated an alternative programming model. It also explains the semantics of Transactional Memory and different design strategies to build a Transactional Memory environment.

2.1 Parallel Programming Models

Nowadays, the two most popular parallel programming models present different characteristics [14]. The Message Passing model splits the computation among threads, which must explicitly communicate their modifications when they are needed by other threads. This technique is quite complex because programmers must know which data is required in each processor and where is this data, and also they have to introduce the communication and synchronization needed to get the data explicitly. Every time that a processor requests remote data, it requires communication with the data owner. The main Message Passing programming language is Message Passing Interface (MPI) [1].

In the Shared Memory programming model [39], concurrent threads share the same memory space and can access the same variables, what allows implicit communication through variables stored in the shared address space. Shared Memory is taking advantage respect to Message Passing because it doesn’t require explicit communication, which usually introduces an important overhead in the computation. Traditional programming languages use the Shared Memory model to support multi-threaded applications. OpenMP [2] is the most popular Shared Memory interface that provides compiler directives to expose the underlying parallelism to the programmer. Java Multithreading [39] and C Pthreads [37] decompose the application to different threads that share data. The code regions, which are manually
identified as parallel, are automatically split into the appropriate number of threads and executed in parallel.

Shared memory allows threads to modify the same memory location concurrently, which may violate the sequential consistency assumption [25] if no explicit ordering is specified in the program. This may affect the correctness of the program if the programmer assumes sequential consistency. Critical sections are program regions where different threads interact with shared memory locations. In order to guarantee that threads behave sequentially, these locations must only be accessed by one thread at a time, disallowing concurrency on shared variables. Hence, Shared Memory model requires synchronization mechanisms to provide mutual exclusion among critical sections.

2.2 Blocking Synchronization Problems

Blocking synchronization is the mechanism used in most Shared Memory programming models to serialize critical sections. Serialization is provided by special flags, known as locks, which indicate if a critical section is being executed by any thread. Every time that a critical section is entered, the thread must acquire a lock to be able to execute the critical code. If the lock is already acquired by another thread, this thread has to wait until the lock owner releases it, at the end of its critical region execution [39].

The example of Figure 2.1 presents a function that updates an inscription table and prints the result. First, it searches the data in a software structure exclusively to obtain the data needed to create the inscription. Isolation is required to avoid updates in the structure during the search. Once it is created, a table is updated and the result is printed. As the printer needs the result of the update, both operations are done together atomically. No changes are allowed in the process. Locks A and B are needed to preserve the atomicity in these critical sections.

Several threads may reach simultaneously a critical section, which means that they will try to acquire the lock at the same time. Systems need to ensure that only one thread acquires the lock, blocking the other requests. Most systems implement locks in software, using atomic instructions provided by the Instruction Set Architecture (ISA), such as Test-and-set (T&S), Compare and Swap (CAS) or Load-Linked Conditional-Store (LL-CS) [14].

However, locks have well-known programmability and performance problems that make parallel programming too complicated for the average developer [50]. The programmer has to choose between the easy-to-use coarse-grain locking and the scalable fine-grain locking, although both approaches have disadvantages. On the one hand, coarse-grain locking defines big critical sections that can only be accessed by a single lock. This locking policy eases the programmer’s work, because he/she does not need to worry about program correctness, as it is assured that critical sections are serialized. Nevertheless, this serialization causes a loss in performance, because most of the time threads are blocked trying to acquire a lock.
On the other hand, fine-grain locking provides more performance, but programmers must deal with several issues that complicate their task. Fine-grain locking defines lots of small critical sections that are accessed using different locks, which allows concurrency between different critical sections. Identifying critical sections is a complex task because it must be assured that other threads do not access the same memory locations in their critical sections.

Figure 2.2 shows two codes that use different grain strategies to implement the function presented in Figure 2.1. The code in the left uses coarse-grain locking, acquiring a single lock to ensure mutual exclusion into a big critical section, which results to serialization with other functions that try to access the same data. The code in the right uses multiple locks to protect different critical sections, increasing the program complexity.

The programmer has also to avoid deadlocks, which appear when a thread that has acquired a lock A tries to acquire another lock B, but this lock has been acquired by another thread, so the thread is blocked until the lock is released. Later, the thread that owns lock B tries to acquire lock A, but it fails because the first thread is its owner. Then, both threads are blocked waiting for locks that will never be released, which stops the threads and prevents the application to finish. Removing deadlocks is not easy, because they might appear only in some executions of the program. This fact makes debugging extremely difficult, but it is necessary to ensure program correctness.

Figure 2.3 presents how the code presented in Figure 2.2 experiments a deadlock. Function update_inscription, which prints the table and updates the table with an inscription, is run concurrently with the fine-grain locking version of create_inscription. Create_inscription acquires lock B to update the table, whereas update_inscription acquires lock C to print the previous inscription. This causes a deadlock, because the create_inscription cannot acquire lock C to print the inscription and update_inscription cannot update the table as lock B is owned in

```
create_inscription(name){
  acquire(lockA);
  data = search_data(name);
  release(lockA);
  insc = create_inscription(data);
  acquire(lockB);
  update_inscription(table, insc);
  print_inscription();
  release(lockB);
}
```

Figure 2.1: Example of a lock-based program
2. Transactional Memory

Figure 2.2: Coarse-grain locking vs Fine-grain locking

create_inscription.

Figure 2.3: Deadlock

Fine-grain locking also presents performance problems that discourage its usage. As threads have several critical sections, many locks are required, which introduces an overhead because the amount of atomic instructions increases. Moreover, having lots of mutually exclusive sections might lead to pathological situations that degrade program performance. One such situation is convoying, which appears when a thread is de-scheduled while it is executing a critical region, causing starvation on other threads that are blocked waiting for that lock. Another non-desirable lock behavior is priority inversion, produced when low-priority threads acquire a lock when other threads with higher priority have been waiting for it.

Furthermore, lock-based code might produce other stability problems. It cannot
be composed because it might cause deadlock, and it is not robust to some system failures, introducing deadlocks when a thread ends in the middle of a critical section by some unexpected interruption. As locks must be explicitly released, other threads that are waiting the lock will never finish.

2.3 Transactional Memory Model

Transactional Memory [22] proposes a lock-free Shared Memory programming model where shared data is simultaneously accessed by different transactions, a concept borrowed from databases. A transaction [17] is a finite sequence of memory reads and writes executed by a single thread. Lock-freedom supplies non-blocking synchronization, allowing concurrency among shared data accesses, and wait-freedom, which ensures that any process completes any operation in a finite number of steps in spite of individual delays.

Transactions are atomic, imposing that all of their memory changes are made visible together, and isolated, guaranteeing that shared data is not accessed concurrently by other transactions. Moreover, they are consistent and durable, so, once they are performed, they appear in sequential order and are forever persistent.

In Figure 2.4 it can be seen how our previous example is modified to use Transactional Memory, removing lock operations. The right code corresponds to the transaction’s body, which searches a name in a global array and updates the array values if the name exists or, otherwise, generates a new entry.

![Figure 2.4: Transactional Memory program](image)

In order to ensure that the transactional properties are transparent to the programmer, the system must provide mechanisms to detect possible conflicts among
2. Transactional Memory

transactions and, in case that two transactions access simultaneously the same data, the system has to abort one of them, performing rolling-back actions to undo modifications introduced by this transaction and re-executing it from the beginning. A transaction commits when it ends and all its memory changes are visible to other threads. These mechanisms ease the programmer’s work, who only has to identify computation on shared data and encapsulate it in a transaction. The programmer does not need to worry about deadlocks because transactions are never blocked.

Transactional Memory also avoids other problems associated with locks [24]. Wait-freedom reduces starvation and removes deadlocks produced from failures. Hence, Transactional Memory is fault tolerant because any unexpected interruption aborts the transaction, allowing other threads to continue and ensuring that this transaction will be executed later. The priority inversion and convoying pathologies can disappear because transactions are able to be aborted if they have low priority or if the thread is de-scheduled. Transactions can be composed because transactions do not generate deadlocks.

Figure 2.5 presents the body of three transactions that are executed simultaneously. The first and the second transactions search for a name that is not present in the array, so both transactions try to create a new location in the global array, which generates a conflict. This conflict provokes the abort of the second transac-

![Figure 2.5: Transaction’s behavior](image-url)
tion, triggering the roll-back mechanism. Then the transaction restarts from the beginning until it commits. The third transaction also accesses the same array, but it does not interfere with the other transactions, so it can be executed concurrently without violating the isolation principle.

Non-blocking synchronization allows optimistic accesses to shared data, which implies more parallelism. Transactions are only serialized when they conflict, so they achieve fine-grain locking performance automatically. In some high-contingency situations, Transactional Memory outperforms fine-grain locking because it does not have the overhead of lock acquisition.

Transactional Memory solves programming and stability problems related to locks without losing fine-locking performance. However, building a Transactional Memory environment is quite complex. For example, each transaction must track its Read-Write set and keep its speculative state to help conflict detection and roll-back mechanisms. Other features are required to perform commit actions or conflict resolution policies.

2.4 Transactional Memory environments

Several approaches have been suggested for building a transactional environment, each one with its own philosophy and techniques. This section gives some background in Transactional Memory environments, exposing the main differences among methodologies and the advantages and disadvantages of each model.

2.4.1 Software Transactional Memory

Shavit et al. [46] proposed Software Transactional Memory (STM), an application programming interface that supports transactions for synchronizing shared data accesses. This approach acquires exclusive ownership on accessed locations and tracks the old memory values using software data structures and Load-Linked/Store-Conditional operations. If a transaction fails to acquire the ownership, it changes its status to failure and restarts later. In case it succeeds, the new memory state is calculated using new memory values, and afterwards the ownership of the memory location is released.

Notice that software determines how the memory locations are owned and decides which transactions fail, allowing flexibility and fair policies. This simple scheme helps the development of efficient STM, built using locks and fast data structures to perform conflict detection and version management. For example, Object-Based STM [5] guarantees atomic isolation in a higher abstraction level, thus removing false positives provoked by a wrong granularity policy.

STM offers a transactional API on current machines, as it does not need any hardware support because transactional state is tracked by software. It also permits portability of transactional programs, because STM does not depend on machine
specific features. Moreover, contrary to simple hardware schemes, STM can deal with transactions of any duration, surviving context switching or page faults, and can handle transactions of any size, because transactions do not overflow resources. Another important feature of STM is software conflict management, which offers flexible policies to reduce starvation and conflicts among transactions.

Although STM have been improved in the last years [21], their performance is still far away from hardware approaches. Memory operations must detect conflicts walking atomically the set of memory locations accessed by other transactions (called summary access or Read-Write set), and commit or abort mechanisms have to perform in the same way. Accesses to the Read-Write set and transactional updates must be explicitly done by software, which increases the latency of memory accesses. Transactional state also has to be initialized at the beginning of a transaction, which increments STM overhead.

These are the main reasons why STMs perform poorly compared to applications that use locks, and this fact encouraged computer architects to apply aggressive strategies to achieve competitive performance. Adding hardware support in the processors helps reducing memory latency accesses and updating transactional state.

2.4.2 Hardware Transactional Memory

Herlihy et al. [22] introduced Hardware Transactional Memory (HTM) as a new multiprocessor architecture intended to make lock-free mechanisms as efficient as conventional techniques based on mutual exclusion. Their design uses typical cache management and coherence protocols on non-transactional operations, and provides extra instructions for transactional accesses, commit actions and state validation. They also proposed hardware transactional support restricted to primary caches and the instructions needed to communicate with them.

The hardware support consists of an associative transactional cache that tracks the lines that are accessed by a transaction with both their old and new values. Conflict detection is performed snooping other processor’s transactional caches, given that they track the memory locations accessed by a transaction. In case of conflict, it aborts the transaction that does not own the line, discarding the speculatively modified lines and reestablishing the old values. If a transaction commits, the new values are committed and the old values are discarded, releasing the Read-Write set.

There are several proposals that improve this design [19][34][10][29][47]. As part of this Master Thesis we analyze the characteristics of these proposals. We defer the discussion of these proposals to Chapter 3. Hardware Transactional Memory applications usually perform at least as well as their respective lock implementation, independently of their particular implementation. However, HTM designs have presented some drawbacks that discourage their implementation in real processors.

The first proposals of HTM schemes tracked transactional state in caches, which bound the size of transactions and put the responsibility of not overflowing trans-
actional resources on the programmer. This fact reduces the portability of transactional programs, because they depend on hardware implementations. However, some unbounded HTM systems have recently been proposed to decouple conflict detection and version management from caches \cite{4}\cite{40}\cite{56}, allowing transactions of any size and duration that are able to survive context switches, page faults, and overflows of transactional resources.

These HTMs store transactional values in software structures, so they require an interface between the hardware and the software. However, these software structures are complex, and may hurt performance if they are frequently accessed. Minh et al. \cite{13} experimented that common-case transactions are short, so most of the transactions do not overflow hardware resources. As some of these HTM only use software mechanisms when transactions overflow hardware structures, they can avoid the overhead of using software mechanisms.

### 2.4.3 Hybrid Transactional Memory

In order to achieve both best-effort performance and flexibility, some designers have proposed Hybrid Transactional Memory (HyTM) \cite{15}\cite{24}\cite{27}, a scheme that combines HTM when it is available and STM when a transaction overflows the resources or when it interacts with external events (page faults, system calls or context switching).

HyTM offers portability when it is executed in software, and performs similar than current HTM because most of the time it is executed in hardware mode, given that common-case transactions are short and do not suffer from unwilling external events. Moreover, HyTM simplifies the idea of unbounded HTM, because it easily handles critical transactions by software, instead of using complex hardware mechanisms. That is the reason why some classical HTM schemes, like TCC with XTM \cite{12}, use software interfaces for transactions that cannot progress on hardware.

However, HyTMs have to deal with some problems that arise when the system runs transactions on different modes. Both hardware and software have to be aware of all transactional states that are kept on other processors, which implies communication between transactions that are in software and hardware mode. This fact slows down hardware transactions when they try to access to the Read-Write set of a software transaction, or when the software monitors hardware resources to detect possible conflicts.

### 2.4.4 Hardware-Accelerated Transactional Memory

Hardware-Accelerated Transactional Memory (HATM) consists on a Software Transactional Memory that uses few hardware resources to accelerate its execution. These changes do not considerably increase the cost and the complexity of the system, although the software can take important benefits of their incorporation.
One of the worse transactional overheads of the classical STMs is conflict detection, because it is required in each memory access, what implies multiple looks through the software Read-Write set. To reduce the summary access, Saha et al. [42] added mark bits in the cache to inform the software if a line has been written by a processor’s transaction or if there is any conflict in the lines that the software has marked. Shriraman et al. [48] introduced extra transactional states in the cache due to modify the coherence protocol to detect conflicts among transactions. Minh et al. [8] used hardware signatures to track the Read-Write set for pending transactions and perform conflict detection among threads.

HATMs are a good proposal to introduce HTM in the market, because they are cheap and do not require much complexity. However, they still need a STM and an interface to communicate with the hardware support, what slows down transactional execution.
Chapter 3

Hardware Transactional Memory

Hardware Transactional Memory minimizes transactional overheads by aggressive silicon policies, introducing hardware support to preserve the transactional state. There are several HTM proposals, each one with its own benefits and overheads. Instead of explaining how current HTM designs work, this chapter abstracts the main mechanisms needed to build a safe transactional environment and explains the strategies that can be followed to implement such mechanisms. For each mechanism presented, we refer to the state of the art of HTM systems that utilize the mechanism.

There are several ways to classify HTM mechanisms. Bobba et al. [7] pointed that HTM systems reflect choices from three different transactional mechanisms: conflict detection, version management, and conflict resolution. Hill et al. [23] categorized current HTM systems by their access summary mechanism and how they perform nested transactions. We decided to merge both categories and add a section in order to explain how current proposals deal with some OS operations inside a transaction.

3.1 Access summary

Access summary is the mechanism that tracks the memory locations that have been accessed by a transaction. This set of locations, commonly known as Read-Write set, is needed to detect conflicts between transactions, and it can also help to select the cache lines that have been modified by a transaction, in order to accelerate commit or roll-back mechanisms.

Each processor has its own Read-Write set, and it is updated when a transactional memory access finishes. Then, depending on the access type, the address of the memory location is added in the read or the write set. Some systems track read accesses in a different set than write accesses [19], but other system track them together to accelerate conflict detection [56]. These sets are cleared when a transaction commits or aborts.

There are some decisions that have to be taken when the access summary is designed. First, we must decide the granularity that the system is going to use.
Cache line granularity tracks if a cache block has been read or written, whereas word granularity only marks those words used by the transaction. Word granularity performs better because it removes false conflicts between transactions that access to different words that are in the same line [33], but it increases the hardware cost and the complexity needed to implement the Read-Write set, so most of the proposals use line granularity [34][24][48].

The second important issue is deciding where the Read-Write set is kept. It can be kept in the cache, adding RW bits per line (or word), which are set to true if the transaction accesses the line (or word) [42]. The Read-Write set can also be stored in other hardware structures [56] or in software [4], what increases the complexity of conflict detection. The problem of using cache or hardware resources is that they are finite, so the system must provide mechanisms to track the access summary when resources are overflowed.

Finally, the third choice is related to what information is stored in the Read-Write set. The transactional access summary can hold precisely each accessed line or word address (which are stored in general in the cache), or it can just keep imprecise information of these addresses. Due to this, Ceze et al. [10] introduced signatures to reduce the cost and the complexity of the operations that interact with access summary. A signature is a set of addresses that are collected in a simple hardware structure. Bulk operations are hash functions implemented in hardware that are applied to addresses that try to access the signature. When a processor wants to introduce an address in the signature, it applies the hash functions on the address. A set of bits of the signature are selected and marked, adding the address in the access summary.

Transactions can also ask for signatures from other transactions, usually to perform conflict detection. Given an address from a remote processor, this mechanism applies the same hash function to it, selecting some bits of the signatures. If all of these bits are set to true, then the system determines that this address is in the access summary. Notice that multiple sets of addresses can map into the same signature.

Signature representation has aliasing if two different addresses are mapped in the same signature bits, which can hurt performance by detecting false conflicts (false positives), but not correctness, because every address is mapped in the same signature bits. Multiple parallel Bloom filters can be used to reduce the penalty of these false positives [43].

Signatures reduce the number of bits needed to track the Read-Write set. LogTM-SE [56] uses Bloom filters and signatures to decouple access summary from caches. This fact, combined with a software log that stores old values, allows LogTM-SE to remove transactional information from caches. FlexTM [47] also use signatures to track the access summary.

A recent design based on token coherence [30], called TokenTM [6], proposes a hybrid hardware/software structure to remove false positives. Each memory location has some tokens assigned and each fetch of a line acquire one of them. On the other
hand, stores acquire all the tokens together. When a token can not be acquired, then a conflict is detected. Tokens are released when a transaction commits or aborts.

The top scheme from Figure 3.1 shows a cache-based access summary, which stores the accessed memory location in cache RW bits, with word granularity. This configuration avoids false positives. An extra bit informs about cache evictions, which can imply an access to an overflow structure that records the access summary elsewhere. Conflicts are detected when an address is present in the cache and the word has been written before. The bottom figure corresponds to a signature decoupled from cache. This signature tracks addresses at line-granularity, and may have false positives because multiple lines can mark the same signature bits. A conflict is detected when all the bits selected by the hash functions are set to one.

3.2 Version Management

Version management is the mechanism that decides how and where the modifications introduced by transactions are stored. Transactions are atomic and durable, which means that, if a transaction fails, all of its changes are discarded, and, if it commits, the modifications are visible at the same time to all processors irreversibly.

The model proposed by Herlihy et al. [22] tracked transactional modifications in a separate processor cache that contained old and new values that could only be accessed by the owner processor. In case of commit, the updated data became visible to everyone, and in case of abort, the system restores the old data, discarding transactional changes.

However, this technique is quite expensive, because it needs to track both new and old transactional data, and transactions cannot survive cache overflows. Several other designers propose hardware schemes to reduce the hardware required for version management without losing overall performance [34][40][47]. Moreover, some of these designers explain different techniques that allow unbounded transactions, independently of their size and duration.
3. Hardware Transactional Memory

3.2.1 Lazy Version Management

Lazy version management [7] leaves the old values in memory, storing the new values in other memory locations or in transactional buffers. This technique accelerates abort recovery because the memory hierarchy doesn’t need to be restored and transactional mechanisms are invalidated instantaneously. This is good for releasing data and eliminating the conflicts that involve transactions during their abort process, but usually data movement is required on commit, which increases transactional time.

Transactional Coherence and Consistency (TCC) [19], an HTM design that only allows transactional code, uses first level caches to perform lazy version management. New values are buffered locally, while a second level cache, which is shared among processors, keeps the old values. When a transaction commits, it sends all transactional modifications to the second level cache, making them visible to all the processors. Aborts invalidate all the first level cache lines, which has almost no penalty.

Programmable Data Isolation (PDI) is the mechanism that Rochester proposed in RTM [49] to perform lazy version management by utilizing the buffer capabilities of private caches. RTM differs from TCC in that it allow non-transactional code, which implies that the cache coherence protocol may interfere with transactional state. For that reason, RTM adds two transactional states (TI for reads and TMI for writes) to the typical MESI protocol, to track the lines used by a transaction and ensure isolation and consistency. The lines only leave these states when a transaction commits, moving TMI lines to Modified, or when a transaction aborts, moving both TMI and TI states to Invalid.

Rock TM [29] also performs lazy version management, but it is not clear how it is going to be implemented. It seems that Rock will have a finite buffer that will store transactional writes, which will appear in the memory system when a transaction commits.

These lazy schemes require finite hardware resources, which limits transactional size. Other designs [4][40][47] propose to keep version management in software structures in case of overflows. That is quite expensive, because, each time that there is a transactional memory access, the system invokes a routine that walks the software structure trying to find the requested line. This process can be accelerated using Bloom Filters, but the overhead of these systems is still considerable.

3.2.2 Eager Version Management

Eager version management stores new values in memory, whereas old values are kept elsewhere. This approach makes commits faster, because transactional writes are in place, but slows abort recovery, which increases the overhead in applications with high contention. Transactional Memories that use eager version management are known as Eager Transactional Memories.
LogTM [34] performs eager version management by storing old values in a software log. This log is structured like a stack, and it is updated the first time that a transactional line is written, pushing the old line to the top of the stack and increasing the stack pointer. When a transaction aborts, the roll-back mechanism activates a software handler that restores the old values. This handler recovers the log in reverse order, swapping transactional modifications with the old log lines. Multiple copies of a line can be found in the log. This does not affect correctness because the oldest value is the one preserved when roll-back finishes. However, to reduce the time spent in abort recovering and writing in the software log, a hardware structure can remember if the line has been written in the log before, avoiding recurrences in the log.

Figure 3.2 presents the schedule of a transactional application that executes two different transactions in a Eager Transactional Memory environment, concretely LogTM. Transaction 1 inserts an element in the end of the list, whereas transaction 2 inserts it at the beginning. LogTM performs eager conflict detection (detecting conflicts when they are produced), eager version management (avoiding commit cycles but introducing roll-back mechanism to undo changes), and a requester-stall policy with backoff (stalling a transaction when a conflict is detected, aborting a transaction when a cyclical dependency among stalled transactions is detected, and performing a backoff to eliminate livelocks when multiple aborts exist). Processor 1 stalls when it tries to write \texttt{size}, a variable read by processor 2. Processor 3 stalls when it tries to read \texttt{table[1]}, modified by processor 3. Later, processor 2 updates \texttt{size}, but it has been read before in processor 3, which produces a cyclical dependency and the abort of the younger transaction. After the abort and the backoff, processor 3 retries the aborted transaction. No commit actions are required.

Figure 3.2: Eager Transactional Memory behavior
3.3 Conflict Detection

Conflict detection is the mechanism needed to detect inconsistencies among transactions so as to preserve atomicity and isolation. One way to do this is by broadcasting memory requests and comparing the address requested with the access summary of other processors [22]. However, this approach considerably increments the amount of traffic of the interconnection network and the latency of each memory access, which increases transactional time. For that reason it is necessary to build more complex transactional protocols.

3.3.1 Eager Conflict Detection

Eager conflict detection [34] keeps transactions in a consistent state by detecting conflicts when a memory operation is performed. Then, if there is no conflict, the system is sure that this location has not been modified by other transactions, so a processor can update its summary access. Transactional coherency is easy to implement when the transactional lines are placed in the first level of the memory hierarchy, but the complexity increases when the lines are evicted from the caches.

LogTM uses a quite sophisticated coherency protocol that ensures transactional isolation even if transactional lines are evicted from the caches. Basically, when a line that has been accessed inside a transaction is evicted from the cache, it is put to a sticky state in the directory (placed at the second level of the memory hierarchy). Then, the directory assigns the ownership of the line to the processor that has evicted it. If a sticky line is requested, the directory sends a message to the owner to force the checking of the access summary. This guarantees that all requests to lines accessed by a transaction will be checked even though the data has been evicted from the cache.

The LogTM coherency protocol is the protocol that we have used in the HTM schemes that are explained in the next chapters. This is the reason why it is explained in detail in Annex 1.

Rochester University presented another interesting transactional coherency protocol, called RTM [49]. This protocol updates L2 directory states the first time that a transaction requests a line. When the line is served, each processor keeps the line in a transactional state. RTM only differs from LogTM in the fact that it allows inconsistencies among transactions, so the L2 directory must support multiple owners of a transactional line. Like LogTM, RTM performs eager conflict detection, but software decides when a transaction aborts.

3.3.2 Lazy Conflict Detection

Lazy conflict detection delays the detection of conflicts until commit time, allowing optimistic concurrency among transactions. This might introduce inconsistencies if a violation occurs, which implies lazy version management to keep the previous
transactional state in the memory hierarchy. Transactional Memories that use lazy conflict detection are commonly called Lazy Transactional Memories.

TCC [18] performs lazy conflict detection by broadcasting the Write Set to other transactions when a transaction commits. Then, non-committed transactions snoop the addresses and compare each one with their own access summary. In case of collision, a conflict is detected, and non-committed transactions are aborted. To ensure correctness, only one transaction can be committed at a time, what might cause serialization. There are some proposals to accelerate the commit phase [11][33].

Figure 3.3 presents the transactional schedule of the previous example when a Lazy Transactional Memory is used. It performs lazy version management, so aborts discard the changes automatically, but it needs actions to store the values during its commit phase. Lazy conflict detection aborts transactions to eliminate inconsistencies at commit time. An arbiter is also needed to serialize transactions, because multiple transactions cannot commit simultaneously. The arbiter is centralized, which increases commit latency. In this example, processor 1 commits first, and it restarts processor 2 and 3 because both have read `size`, a variable written by 1. Then, processor 2 and 3 restarts, and both reach the commit phase, but only processor 2 can commit, so processor 3 waits. During this time, processor 3 detects a conflict and aborts.

FlexTM [47] is a variation of RTM that aims to detect conflicts in both eager and lazy mode. Internally FlexTM detects conflicts eagerly always, but its speculative version management system allows inconsistencies among transactions, so conflict resolution can be delayed until commit time, emulating lazy conflict detection.

Lazy conflict detection guarantees concurrency among transactions, avoiding continuous mutual aborts that generate livelocks between transactions. Optimistic
Hardware Transactional Memory

concurrency also reduces the overhead of keeping coherency in each memory access. On the other hand, eager conflict detection is able to catch conflicts when they are produced, resolving the violation rapidly and giving the opportunity to resolve the conflict as soon as possible.

3.4 Conflict Resolution

Conflict resolution, also known as conflict management, is the mechanism that determines how a conflict will be resolved once it is detected. This decision can be determined at the moment that the conflict occurs (eager conflict resolution) to preserve the consistency among transactions, or later (lazy conflict resolution), allowing flexibility in the resolution policy.

Eager conflict resolution is the most widespread technique, because it ensures consistency among transactions. However, there are several policies that have an important impact in performance, because pathological situations may appear when resolution decisions are taken [7].

As we described in previous sections, TCC performs lazy conflict detection, which means that it requires serialization among transactions at commit phase. The resolution policy determines that if some non-committed transaction has accessed a value modified by the committing transaction, it must abort in order to preserve isolation. This policy is quite drastic, because it discards useful work, and can starve older transactions and produce convoying. Some proposals intend to reduce these pathologies [52] using fairly policies.

LogTM detects conflicts eagerly, which allows flexible resolution policies. LogTM employs a requester-stalls policy, which means that a transaction that wants to modify a line that has been read by another transaction or intends to read a line modified by a remote transaction must stall until the owner of the line commits. A cycle-detection mechanism is needed to detect deadlocks between stalled transactions and, in that case, abort the youngest to balance the work.

This resolution policy may introduce livelocks if two transactions abort each other repetitively. For that reason, a backoff is needed after the abort recovery to separate them and ensure progress. Other policies (like timestamp) can be used to remove backoff.

LTM [4] is a simplification of UTM that uses requester-wins Conflict Resolution policy. This policy aborts the transaction that owns a line when some other transaction requests it. The requester wins policy can introduce livelocks, because multiple transactions may request the same memory location, which produces recurrent aborts.

FlexTM [47] has recently decoupled conflict detection and conflict resolution. Conflicts are detected eagerly, but FlexTM is able to postpone conflict resolution until commit time. Conflicts are stored in three hardware tables per processor, called Conflict Summary Tables, which contains one bit for each other processor in
3. Hardware Transactional Memory

the system. These tables track if a local read conflicts with a remote write (R-W table) or if a local write conflicts with a remote read (W-R table) or write (W-W). In each coherence request, the controller checks the signatures, updates the tables and sends information in its response to update remote tables. This mechanism allows flexible resolution policies that simulate an efficient lazy environment.

Figure 3.4 recreates a flexible conflict resolution behavior, similar to FlexTM. It performs eager conflict detection and lazy version management, although some extra actions have to be performing during the aborts. Lazy conflict resolution allows transactions to abort at commit time. This removes the commit arbiter and the eager backoff, and a flexible software policy decides which transaction aborts. In this example, processor 1 is the first to commit, but the software policy decides to abort it to ensure the progress of an older transaction on processor 2.

![Figure 3.4: Flexible Conflict Resolution behavior](image)

3.5 Nested Transactions

A nested transaction [36] is a transaction whose execution is properly contained in the execution of another transaction. As the inner transaction is atomic, isolated and does not block, the programmer does not need to care about its progress or correctness. This allows using transparent code, such as modular code or libraries, because it is composed without creating race conditions.

There are several ways to implement nested transactions in Hardware Transactional Memories. Flat nesting treats all the transactional code as a single atomic block limited by the outer transaction. Aborting the inner transaction causes the abort of the outer transaction, but committing the inner one has no effect until the outer transaction commits, at which point the inner transaction changes become visible to other threads. The outer transaction sees the modifications made by the inner transaction when the latter commits.
Flat nesting is easy to implement, as it does not need important changes in HTM designs. Inner transactions share the access summary, and version management support with the outermost one, so no extra support is required when an inner transaction starts. However, flattened transactions subvert program composition, since an abort in an inner transaction terminates all surrounding transactions.

There are two alternatives to improve flat nesting performance: closed and open nesting. Closed nesting works the same as flat nesting with the exception that aborting an inner transaction does not abort the outer one. Commits of inner transactions are not visible until the outer ones commit, like in flat nesting. Closed transactions require more transactional actions and hardware support. The access summary must be tracked separately to detect which nesting level provokes the conflict, and, in the same way, version management has to identify the modifications that each transaction has introduced to support partial roll-backs.

McDonald et al. [32] proposed two possible implementations for closed transaction nesting in TCC. The first design replicates RW bits for each nested level, but only tracks the innermost modifications. The oldest data is recorded in a log. When an abort occurs, it detects the conflict nesting level using several RW bits per line. Then, all the changes introduced by this transaction (and its inner transactions) are discarded, replacing the cache values with the ones stored in the log. The second proposal tracks nesting level modifications in different cache lines, which reduces the overhead of partial aborts, but requires a level identifier per line and multiple copies of modified values.

LogTM-SE [35] performs closed nesting, tracking nesting access summary in different signatures. When a nested transaction starts, a new log header is introduced to isolate its log space from the parent transactions. If a nested transaction aborts, the software handler restores the state of the conflicting transaction, performing a partial roll-back. This is done by walking the log until the header of the nesting transaction is found. Old values from outer transactions are still in the log, so the nesting transaction is re-executed without aborting the parent.

The second design alternative is open nesting. The only difference between open and a closed nesting is that open nesting makes the changes of the nested transaction visible to all before the outer transaction commit. This breaks transaction isolation. This property reduces transactional overheads, eliminating unnecessary conflicts, but introduces inconsistencies among transactions that might affect the program correctness. This happens when an open nested transaction commits and, later, its parent must abort. As open nesting discards transactional state when a nested transaction commits, some transactional modifications remain after the abort recovery. Due to this, some compensation code is needed when a nested transaction aborts to undo open transaction’s changes.

Figure 3.5 shows three different ways to implement nested transactions. Flat nesting aborts the outer transaction, retrying all the inner transactions. Closed nesting determines the last conflicting level, aborting until that level and restarting it. In the example, transaction 3 causes the conflict and it is the only one restarted.
Open nesting commits the inner changes immediately, removing possible conflicts (Open nested 1), but in some executions that abort the outer transaction require user compensable code (Open nested 2) to undo the changes of the open transaction 3.

![Figure 3.5: Nested Transactions](image)

### 3.6 OS Support for HTM

For transactional memory to become the new standard for multi-threaded programming, it must efficiently support system calls, I/O operations, context switching and paging, especially if such events occur during the execution of a transaction.

System calls are actions executed in kernel mode and are necessary when the application needs to interact with the system. It is common to find system calls inside a transaction (for example, memory allocation), which may interfere with desirable atomic behavior. System calls must be executed atomically, and they might modify system state. Moreover, as they are executed in kernel mode, user roll-back code cannot be applied, so, once a system call starts, it must finish. Other alternatives might leave the system in an inconsistent state. For these reasons, system calls are treated as non-transactional code.
Log-TM uses escape actions to resolve any memory request in a system call. These actions do not require transactional coherency, so they behave like a non-transactional memory request. Although an abort request might appear during a system call, roll-back actions are not performed until the system call finishes. The changes introduced by the system call are not rolled-back.

MetaTM [41] an active transaction using system primitives. These primitives deactivate local transactional updates, allowing transactional threads to execute non-transactional code. However, other transactions must abort if they conflict with the suspended transaction. With this mechanism, updates to memory are visible immediately, and transactions are not rolled back during the system call. This mechanism also can be used to resolve I/O operations. TCC uses a similar interface to execute system calls.

Context switching and paging are more complex to solve than system calls and I/O. An easy way to address context switching and paging is to abort the current transaction, perform the OS action and then restart it. However, this mechanism does not work with transactions that are longer than an OS scheduling quantum. Such transactions would result in a live-lock situation, since they would continuously abort and restart on every context switch. These occasions require a mechanism to virtualize transactions [40].
Chapter 4

Hardware Approaches for Transactional Memory

This chapter presents different Transactional Memory models with unbounded hardware support, describing LogTM as a baseline HTM and discussing other approaches to improve its performance. Some of these approaches have been proposed before while others are novel to this study. The following sections explain the workings of these models, what the hardware that they require is, and how this support is used in a transactional environment.

4.1 LogTM

LogTM [34] is an eager version management HTM that stores old transactional values in a private software log. This eliminates commit actions because all the changes introduced by transactions are in place, but requires a software handler to restore transactional state in case of abort. LogTM performs eager conflict detection using the coherency protocol described in Annex 1. When a transaction requests a cache line that is owned by another transaction (i.e., the latter has written the line), the conflict resolution policy of LogTM stalls the requesting transaction until the owner of the line commits. If there is a cyclical dependency between the transactions, then one of the two is aborted to avoid deadlock. This policy does not ensure progress, so a backoff is needed to remove a cascade of aborts. The behavior of LogTM is described in 3.2.

Later LogTM refinements [35][56][43] minimize the hardware needed to track transactional state. In its first incarnation, LogTM used RW bits in the cache and in the directory to track the access summary, using line granularity. Later, LogTM-SE [56] introduced signatures to decouple RW bits from caches. As this chapter analyses HTM with infinite hardware support, we assume perfect signatures to track the access summary.

However, other features are required to ensure transactional correctness. First, the system needs to roll-back processor state after an abort. This implies undoing
both memory and register changes. Register map table checkpoints [3] can be used to restore processor state. This technique, used in branch misprediction recovery, replicates at every branch the state of the processor in a separated hardware table. On a misprediction, the checkpoint is rapidly restored, allowing the execution of the correct path.

In a Transactional Memory system, a checkpoint is created when a transaction starts. If the system permits nested transactions, the number of checkpoints determines the nesting depth. When a transaction aborts, the processor state is restored by copying the appropriate checkpoint. Checkpoints are discarded during the commit phase. This technique is applied in all the HTM models presented.

A nesting level counter must keep the nesting level of each transaction. This allows the system to determine if it is executing a transaction or not. This counter is incremented when the transaction starts and is decremented when it commits or aborts. A memory access is transactional if the nesting level is higher than zero, which enables the conflict detection. Only escape accesses deactivate conflict detection. A memory access is escaped when it belongs to a system call or when the system is sure that it will not affect program correctness.

The rest of the hardware support required for LogTM depends on the version management strategy. It consists of keeping a copy of older values in a software log. This software log must be initialized when a transaction begins, introducing a header that contains its nesting depth, a pointer to its parent and the current Program Counter (PC). This information determines where the transaction started.

Each transactional store must follow three steps to ensure that the new value is in place and the old value is in the log: (1) the system brings the cache line to the processor if it is not already there, (2) the old data is stored in the log, and (3) the new data is stored in the cache.

To speed-up this process, we assume a LogTM design that is able to read the old line and write the new value in the same cache access. The old line is momentarily stored in a latch, which also contains the modified memory location. Assuming a 64 byte cache line and a 32-bit processor, an undo log entry consists of 64 bytes to keep the old cache line, 26 bits to track its address and 6 bits that identifies it as an undo entry. Notice that a log entry requires 68 bytes, distributed in two different cache lines. This means that log entries are not aligned at cache line boundaries.

Moreover, the system has to keep a log pointer that tracks where the entry is located in the software log. This pointer is an address that is incremented by the entry size. Hence, once the old data is placed in the latch, the system can store the undo entry in the log, moving the latch values plus the identifier to the memory hierarchy. These two movements do not need to check the RW bits of other transactions, because they update a memory section that is private to the thread.

Logging has a non-negligible cost, because it requires several movements to memory. For that reason, two different improvements can be implemented to reduce the time spent logging. First, a table can be used to track those addresses that have
been written before inside this transaction. These lines are present in the log, so there is no necessity to replicate them. If the address is not in the table, a new undo entry is stored in the log, although the log already contains it. That is not a problem, because, if an abort occurs, the log is restored in reverse order. Figure 4.1 shows a diagram of the logging process.

The second improvement tries to reduce the latency of storing into the log. A prefetch request accelerates the log process, bringing the log entries close to the processor. This request can be sent at the same time we update the memory with the new value. Notice that a transactional store could be a long latency operation, because it might need to check the access summary of remote processors.

Let’s focus now on how the software log is used. Obviously, it will be necessary to restore those memory locations that have been modified by an aborted transaction. As transactions are durable, once they commit, their changes are preserved forever. This means that the thread log can be discarded. An easy way to remove it is updating the log pointer with an address that points at the base of the log.

The conflict detection mechanism permits the identification of those transactions that have to be aborted. When a conflict between two transactions is detected, one transaction stalls waiting for the other transaction to commit. However, to avoid cyclical dependencies between stalled transactions that might produce dead-
locks, transactions must inform a centralized cycle-detector when they are stalled. This cycle-detector keeps track of which transactions are stalled, who stalled them and when these transactions began. When a cycle occurs, a transaction timestamp, updated when a transaction begins, determines the younger transaction that participates in the cycle and raises an exception on it.

This exception calls a recovery handler. The recovery handler requires some information kept in the processor, such as the number of retries (to compute the backoff), the conflicting nesting level (to perform partial roll-backs) and the log pointer (to recover old data values). This information is pushed into the thread’s stack, and the PC is updated with the handler address.

The recovery handler is a software routine that walks the log in reverse order and, for each undo entry, stores the old line data at the address kept in the entry. When it arrives to the header of the log, the handler informs the hardware that it can clear the access summary. Then, if the level of the header corresponds to the current nesting level, a software backoff is performed. Otherwise the recovery continues walking the parent’s log. When the backoff is finished, the recovery mechanism returns the control to the hardware, setting the PC to the value stored in the header log.

4.2 Transactional Buffer

LogTM version management mechanism does not require extra storage hardware and no memory movement is performed when a transaction commits. However, two things degrade its performance. First, a software handler has to walk the log to restore the previous state when a transaction aborts. This is a slow process that could be accelerated by hardware. Second, each transactional store might require an update in the software log, which increase the latency of transactional operations. It is important to reduce the latency of transactional operations, because it reduces both transactional time and contention.

This section proposes the incorporation of a hardware buffer to substitute the software log and, at the same time, to add flexibility in the version management and conflict detection policy, allowing eager and lazy approaches. This structure is similar to a gated Store Buffer [54] or the speculative Buffer used in Trips [44]. We will call this structure the Transactional Buffer.

The Transactional Buffer consists of a hardware table that stores a word memory address, the 32-bits value associated with that address, and a valid bit. This buffer can be used to track the old values (Section 4.3) or the new values (Sections 4.4-4.6), and can be implemented using CAM technology [45].

Nested transactions are flattened to simplify the Transactional Buffer logic. This means that, if a nested transaction aborts, the outermost transaction must be re-executed. Flat nesting permits associative searches in the table without worrying
about the nested level. Other implementations might store multiple copies of a memory location, each one associated with a nesting level.

The Transactional Buffer has a cost in complexity, area and power. However, there are several reasons why a Transactional Buffer is useful. First, it simplifies the log process, reducing the latency of storing in the software log. Second, it operates with word granularity; so only modified data is stored in it, consuming less energy in transactional stores. Third, the Transactional Buffer allows a fast roll-back mechanism replacing the software recovery, because all the information needed to modify the transactional state is close to the processor. Moreover, a smarter backoff can be performed instead of the one implemented in the software library.

Finally, summary access is also recorded in the Transactional Buffer at a word granularity, whereas the perfect write signature offers line granularity. This fact restricts false positives. However, we decided to decouple access summary and version management, so a perfect signature is used to track the writes, and word addresses from the Transactional Buffer are used to manage the transactional state.

4.3 Eager Version Management

Eager Version Management Transactional Memory (EVM) stores new values in the memory hierarchy and the old values in the buffer. EVM operates identically to LogTM, but it accelerates its version management mechanism with the Transactional Buffer.

EVM needs a cache which serves the old value to the Transactional Buffer and, simultaneously, stores the new value in the same address. At the same time, the Transactional Buffer checks if this address has been updated in the transaction before. In that case, the old value is discarded, otherwise is stored in the first free entry of the buffer with its address, and the valid bit is set to one. As the Transactional Buffer stores all the transactional modifications, EVM does not require neither accesses to the software log or a look-up table to remove redundant entries in the buffer.

EVM automatically discards the Transactional Buffer when a transaction commits by flushing the valid bits. In the moment that a transaction has to abort, the processor is stopped, and the state is restored by moving the old values from the Transactional Buffer to the memory hierarchy using write requests. As the cache has only one write port, only one request can be sent at a time, and the others wait until it finishes. When the restore process finishes, all the valid bits are flushed. Notice that EVM does not discard its Read-Write set until the abort is completed, so recovery requests don’t need to check other processors’ signatures, which accelerates the process.

Figure 4.2 overviews different operations that involve the Transactional Buffer in EVM. EVM keeps old values in the buffer whereas new values are in the memory hierarchy. A transactional store searches if a location is present in the Transactional
Buffer. If it present, no action is performed in the buffer, otherwise the old value is introduced in the buffer, with its own memory address. A commit flushes the valid bits of all the entries of the Transactional Buffer, whereas an abort stores the values from the buffer to the memory hierarchy.

![Figure 4.2: Eager Version Management using the Transactional Buffer](image)

We also modeled another HTM approach, called Elog, to evaluate the cost of logging in LogTM. This approach stores old values in the software log, but it recovers its transactional state from the Transactional Buffer. Hence, it only experiments the overhead of logging respect to EVM.

### 4.4 Lazy Version Management

Lazy Version Management Transactional Memory (LVM) uses the Transactional Buffer to store speculative transactional values, keeping old values in the memory hierarchy. LVM operates with the other mechanisms of the LogTM, such as conflict detection or conflict resolution, but the version management approach is slightly different, modifying the commit and abort phase, like Rock [29] or BulkTM [10].

LVM keeps transactional stores in the Transactional Buffer, so the old data rests in the memory hierarchy. This fact implies that transactions do not modify the cache until they commit. However, LVM keeps transactions in a coherent state, which means that only a processor can modify a given memory location. For that reason, the same eager conflict detection mechanism as LogTM has to be used.
However, if the address updated is already in the Transactional Buffer, then the transactional coherency request is not needed, because the processor modified it before and has its ownership.

As transactional modifications are stored in the Transactional Buffer, any read request has to check the buffer to get the correct value. This associative search is done in parallel with the cache access, which is discarded if we hit in the buffer. Otherwise, if the line is not present in the L1, a transaction coherence request is required to preserve the coherence of the memory access, which transports that line to the L1.

Figure 4.3 presents how memory accesses, commits and aborts are performed in LVM. LVM keeps new values in the buffer whereas old values are in the memory hierarchy. A transactional load looks at the buffer if the line is present, and if it is it returns the value stored in the buffer. If not, it returns the value from the memory hierarchy. A transactional store searches if a location is present in the Transactional Buffer. If it is present, the value is updated in the buffer, otherwise a new entry is added in the buffer. An abort flushes the valid bits of all the entries of the Transactional Buffer, whereas a commit moves the values from the buffer to the memory hierarchy.

However, there is an exception where a transactional store modifies both the memory hierarchy and the Transactional Buffer. System calls usually modify mem-

Figure 4.3: Lazy Version Management using the Transactional Buffer
ory locations, and these updates must be done in the memory hierarchy. Otherwise, the Operation System does not know where they are stored. Some of these system calls are executed inside a transaction, and must be done atomically. Fortunately, most of these calls (like malloc or free) don’t need to be reversed or allow some compensation code if a transaction aborts. Each store from a system call is marked as privileged store, which permits the hardware to send it to the memory.

LVM performs an instantaneous abort recovery, because it discards the buffer just flushing the valid bits of the Transactional Buffer. This allows the transaction that wins the resolution to immediately get the critical data, because the summary check of the loser is freed when the abort is detected. On the other hand, commit phase requires extra time, because the transaction needs to move its speculative writes from the Transactional Buffer to the memory hierarchy. It is performed similar to the abort phase of the previous section. Notice that, as the transactions are coherent, a transaction that is committing cannot be aborted. The disadvantage is that this commit phase, which is not needed in EVM, delays transaction execution time.

EVM and LVM improve LogTM’s version management, but they keep its conflict detection philosophy. Eager conflict detection could be right for eager version management systems, given that it prevents inconsistencies and maintains isolation among transactions. However, lazy version management allows flexibility in conflict detection policies, because transactions store their speculative state outside the memory hierarchy, so transactional modifications are isolated from other transactions.

Figure 4.4 resumes how EVM and LVM perform eager conflict detection. Basically, they keep the coherency using a MESI transactional protocol, checking the access summary if the line is not present in the cache or if is shared. However, LVM doesn’t request the line to the L2 if it is present in the Transactional Buffer, so it reduces the traffic for evicted lines.

The following sections presents two approaches that differ from LogTM’s eager conflict detection. Both of them perform lazy version management to keep in the memory hierarchy the old values. They use the same transactional protocol with few modifications in order to eliminate or delay some access summary checks to ensure forward progress and reduce network contention.

## 4.5 Lazy Conflict Detection

Lazy Conflict Detection Transactional Memory (LCD) emulates a TCC environment [19], with two main differences. First, TCC doesn’t allow interference between non-transactional and transactional code. In fact, it only permits transactional code. Our system uses a coherency protocol that is able to operate with data that is accessed inside and outside a transaction. Second, TCC uses a write-through policy, where a shared L2 cache contains the consistent state of the committed transactions,
so the L1 cache just keeps the speculative state. Our approach keeps the speculative state in the Transactional Buffer and stores it at commit time in any level of the memory hierarchy, even if it is a private resource of the processor.

LCD delays conflict detection until commit phase. This fact reduces memory accesses latency because they don’t check other transactions’ summary access. Any transactional modification is stored in the Transactional Buffer without any coherency request. Transactional loads ask the memory hierarchy and the Transactional Buffer at the same time. If the line is valid in the private L1 cache or in the Transactional Buffer, no coherency request is needed; otherwise the line is requested from the shared L2. That line might be not updated in the L2, because the system has write-back L1 caches. This is quite common, because that line might belong to non-transactional code, a committed transaction, or a non-committed transaction.

Notice that, in eager conflict detection, the two first alternatives generate a transition from Modified to Shared state; whereas the third alternative generates a conflict. LCD treats that line as a non-transactional line, performing the same transitions and ignoring summary access checks. This fact introduces an inconsistency among transactions, which must be solved later to guarantee isolation among transactions.

Conflicts are detected at commit time. When a transaction reaches its end, it flushes all its modifications through the memory hierarchy, making the changes persistent by sending stores. This means that, once a transaction starts its commit phase, it is consistent, so it cannot be aborted. These stores generate transactional coherence requests that force the transactions to check their access summary, searching for conflicts. If a conflict occurs, the committer wins the conflict and the other transaction has to abort to preserve isolation. Figure 3.3 in the previous chapter resumes LCD behavior.
4. Hardware Approaches for Transactional Memory

TCC broadcasts the write-set, consuming a lot of bandwidth. As our transactional protocol keeps a list of sharers in the L2 directory, our approach generates a selective check. However, sometimes the directory doesn’t know who used a line (for example, when a line is evicted from the L2 cache), so a broadcast is needed. In fact, the L2 directory works as RTM’s Conflict Summary Tables [47], which decouple conflict detection from conflict resolution.

Figure 4.5 shows how LCD performs coherence requests. LCD performs lazy conflict detection in writes. Whilst EVM generates a coherence request in stores when a transactional line is in not present or shared in the L1, LCD just store the transactional value in the Transactional Buffer, without any communication. LCD treats load transactional accesses as non-transactional loads, so they never generate a summary check.

![Figure 4.5: Lazy conflict detection in LCD and HCD](image)

LCD only allows one transaction committing at a time. This ensures progress, because transactions only abort when another transaction commits, so a transaction can only be aborted by the same transaction once. That is the reason why backoff is not needed in LCD. However, Transactional Consistency forbids more than one transaction committing at the same time, and transactions must be serialized in their commit phase. There is a single exception that aborts a transaction even if nobody commits. If a non-transactional memory request accesses transactional data, the transaction must be aborted to guarantee strong isolation between transactional and non-transactional code.

Committer Arbiter is the responsible to decide which transaction must commit. When a transaction reaches its end, it asks the Commit Arbiter for the ownership of a commit token. If the token is free, the Commit Arbiter allows the transaction to commit, keeping the token busy. Transactions free the token when they finish.
its commit phase. If the commit token has been assigned to another processor, the transaction waits until the token is released. In case that multiple transactions wait to commit, the Commit Arbiter distributes the token using a FIFO algorithm. Of course, if a transaction receives an abort signal while it is waiting for the token, this transaction must abort, removing its token request.

Figure 4.6 resumes lazy conflict detection commit phase. LCD asks permission to a global commit arbiter when a transaction commits. It receives the commit token if it can commit, otherwise the transaction has to wait. When the transaction has permission to commit, it sends coherence requests for each line that is not present in the L1 in Exclusive or Modified states. These requests transports the line to its L1 cache and, at the same time, aborts inconsistent transactions that conflict with the committer.

LCD reduces the latency of transactional accesses by removing repeated memory requests and delaying conflict detection. It also reduces the contention generated by transactions, eliminating ping-pong aborts. This fact promotes the elimination of backoff cycles, which increase the execution time.

Although LCD contributes with nice ideas to improve transactional performance, there are other drawbacks that discourage its usage. For example, transactions cannot be stalled when they find a conflict, because they already are in an inconsistent state, and the transaction must be aborted to preserve Transactional Coherence. These transactions must be restarted from the beginning, even if most
4. Hardware Approaches for Transactional Memory

of the previous work doesn’t conflict with any transaction. Moreover, the usage of a centralized arbiter serializes transactions that don’t have conflicts and may introduce starvation if the distribution of the token is not fair. Finally, whilst LVM wastes few cycles to move data from the Transactional Buffer to the memory, LCD’s commit phase takes more time because summary checking is required.

4.6 Hybrid Conflict Detection

Hybrid Conflict Detection Transactional Memory (HCD) tries to mitigate the performance pathologies suffered by LCD. HCD merges both eager and lazy conflict detection, performing eager conflict resolution. It takes benefit from the implementation of the transactional coherence protocol, which always looks at other summary accesses searching for conflicts. Basically, HCD keeps transactional reads in a consistent state, whereas it speculates with transactional writes and their posterior reads.

On one hand, HCD performs lazy conflict detection for transactional writes, using the same mechanism as LCD. This means that stores are speculatively stored in the Transactional Buffer without checking for conflicts. This reduces write latency and network contention. However, HCD also needs the Commit Arbiter and the token policy in order to prevent multiple commits at the same time.

On the other hand, HCD performs eager conflict detection to detect conflicts between transactional loads in one transaction and stores from other transactions. When a read line is not present in the L1 cache, the processor sends a request message to the L2 directory, who holds the lines that have been modified by other processors. If someone has written that line, the writer has to check its write signature in order to detect conflicts. It doesn’t require extra time because the coherence protocol does that by default.

This fact allows keeping the transaction in a pseudo-coherent state, because most transactions load data before writing it. Instead of aborting, a transaction may remain stalled until the conflict disappears. Stalls reduce the number of retries and allow the reuse of previous work. Of course, a previous update may provoke an inconsistency with a committing transaction, which generates an automatic abort, even if the transaction is stalled. However, a backoff is needed after the abort to distribute contention and eliminate livelocks, similar to eager conflict detection HTMs.

Figure 4.5 summarizes how HCD interacts with the Transactional Buffer. HCD performs lazy conflict detection in writes, so it just store the transactional value in the Transactional Buffer, without any communication. Loads generate a summary check in HCD when a line is not present in the Transactional Buffer or in the L1 cache.

HCD mixes Eager and Conflict Detection. Consequently, it has both advantages and disadvantages: it avoids the main pathologies of other HTMs but shares
their principal drawbacks. Transactions increase their latency adding stall, waiting, commit and backoff cycles, but this fact permits HCD to decrement the overheads produced by contention.

Figure 4.7 presents an example of the behavior of HCD, using the Chapter 3 transactional program. Processor 1 and 2 modify the size of the list at the same time, but the shortest one ends first. As the transaction in processor 2 has an inconsistency, it has to abort. Processor 3 detects the conflict the first time that reads the size, and it stalls. Once transaction 1 is committed, the stalled transaction can be restarted. The aborted transaction performs a backoff after the abort, but when it restarts detects a conflict eagerly and stalls until processor’s 3 transaction commits.

![Figure 4.7: Hybrid Conflict Detection Behavior](image-url)
Chapter 5

Bounded Support for HTM Approaches

Chapter 4 discussed different Transactional Memory approaches and the hardware that they require. A Transactional Buffer improves Version Management by keeping in hardware the transactional state, whereas Perfect Signatures record transactional accesses at line granularity. Both of them don’t take into account the size of the transactions because in the previous chapter we supposed idealized hardware support.

However, these hardware structures require lots of silicon and complex logic, having a high cost in area and power. Moreover, some HTM approaches perform associative searches in the Transactional Buffer, which suggests that the Transactional Buffer should contain only a few entries. Nevertheless, the hardware has to be transparent to the programmer, who doesn’t have to worry about possible hardware overflows. Due to this, each HTM approach must be implemented with finite hardware support, but providing mechanisms to allow the execution of transactions of any size.

Current HTM proposals use complex strategies to handle overflows, which slows big transactions and requires extra hardware [4][40]. Other hybrid designs run overflowed transactions in a STM, which delays even more their execution. This chapter proposes several alternatives that permit transactions to survive hardware overflows without adding extra-hardware or paying the price of moving to software.

5.1 LogTM-SE

LogTM-SE [56] decouples transactions from caches using signatures (large bit tables) to store transactional accesses, which permits the elimination of RW bits in the L1 and the sticky states in the L2. Lots of addresses are stored in the signature using hash functions, which might introduce false positives. LogTM-SE requires a signature per nesting level, and they are checked in parallel to detect conflicts. If signatures are not available, they are virtualized. All the HTM approaches that were presented in Chapter 4 use signatures to record transactional accesses.
LogTM-SE, like LogTM, keeps the old transactional state in a software log, so it can support transactions as large as physical memory. LogTM-SE supposes that abors are rare, so it can efficiently handle overflowed transactions without delaying their execution, because transactional loads and stores directly interact with the memory hierarchy. In case that a transaction generates a log larger that physical memory, the transaction can be virtualized [51].

LogTM-SE offers the simplest mechanism to handle overflows, basically because it doesn’t require hardware for Version Management. However, recent studies [16] demonstrate its handicaps when high-contention applications are executed. Our goal is to use the hardware support when it is available and LogTM-SE when resources are exceeded. This fact implies that the system executes different Transactional Memory approaches at the same time. The mode of a transaction determines which Transactional Memory approach it uses. Transactions can switch between modes to handle overflows or to return to the common-case mode.

5.2 Bounded Eager Version Management

EVM is a Hardware Transactional Memory that stores the previous transactional state in a Transactional Buffer to accelerate the recovery mechanism. However, only a few entries can be kept in the hardware given its finite dimensions, and the transactions that overflow it must be executed in a safe mode. We propose two alternatives that use the support provided by LogTM-SE to handle these transactions.

The first alternative consists of using EVM mode in all transactions until one overflows. In that case, the overflowed transaction must abort, because it didn’t keep its state in software. After the abort, the processor sets an overflow flag to true, which implies that the transaction is executed in LogTM-SE mode, storing the old lines in the software log and performing software roll-backs in case that it aborts. Aborts caused by overflows do not require backoff. The overflow flag is set to false when the transaction commits, so the next transaction is executed in EVM mode.

The problem of this proposal is that it aborts all the useful work done by the overflowed transaction. If overflows are not common, the price of the abort is not representative, and might be beneficial in order to avoid starvation, because it permits the progress of small transactions stalled by the large transaction. However, if overflows abound, the price of aborting and retrying is a big liability.

A good alternative to get rid of unnecessary aborts is to keep the software log for all transactions. Then, old values are placed in the hardware until it overflows, which sets the overflow flag to true. When a transaction aborts, it checks this flag to decide if it has to perform the roll-back via hardware or software. The problem of this approach is that it logs all transactions, even the small ones that fit in the Transactional Buffer. Chapter 6 shows that logging has almost negligible cost, so we implemented the logging version of bounded EVM, called BEVM.
Figure 5.1: Hardware support unbounded EVM with logging

Figure 5.1 shows the hardware support needed in BEVM. Stores send the old line in the Transactional Buffer and the software log. An abort is recovered by hardware when the Overflow flag is set to false, otherwise the transactional state is restored using the software log.

5.3 Bounded Lazy Version management

Bounded LVM (BLVM) is less effective in software logging than BEVM because it stores the transactional modifications in the log, and this log has to be accessed each time a memory location is loaded. Some proposals [40][47] accelerate this process using bloom filters and look-up tables, but it increments the complexity of the mechanism and its effectiveness hasn’t been contrasted. Instead, LogTM-SE offers an easy way to survive overflows.

Nevertheless, it is not so easy to move from LVM to LogTM-SE, because their Version Management policy differs. LVM keeps the values previous to the transaction in the memory hierarchy, whereas LogTM-SE stores them in a software log. Hence, the Transactional Buffer and the software log don’t contain the same transactional state, so there is no benefit to log in hardware and software at the same
A simple way to handle overflowed transactions consists of aborting overflowed transactions and re-executing them in LogTM-SE mode. Unnecessary aborts only increment retry cycles, because lazy version management recovers the transactional state immediately. Notice that there is no inconsistency in the usage of eager and lazy version management at the same time, because both modes use eager conflict detection.

Figure 5.2 shows how BLVM handles overflowed transactions. Transaction from processor 2 reads the size of the array and starts some work on it. Transaction 1 tries to modify the size of the array, but it cannot because processor 2 has read it before. The Transactional Buffer is overflowed in the middle of a transaction, what causes an abort in processor 2. This fact allows processor 1 to finish its job. Processor 2 doesn’t require backoff. Processor 3 has aborted a LogTM-SE transaction, so it needs to recover the old state using the software log.

This design is evaluated in chapter 6. Results show that it performs worse than BEVM because it re-executes overflowed transactions in LogTM-SE, what means that all the aborts are recovered using software routines. EVM restores previous transactional state by hardware when a transaction fits in the transactional buffer. Some of the studies that we have done pointed that most of the aborted transactions in eager conflict detection fit in the Transactional Buffer, what accelerates the rollback mechanism in BEVM.

This fact encourage the re-execution of overflowed transactions using BEVM instead of LogTM-SE. Overflowed transactions store the old data both in the software log and in the Transactional Buffer, and they are recovered by hardware when the aborted transaction doesn’t overflow the Transactional Buffer. This design, which...
is called Optimized BLVM (OBLVM), can be also applied in the bounded version of HCD and LCD to accelerate the execution of overflowed transactions.

5.4 Bounded Hybrid Conflict Detection

HCD uses lazy version management to maintain the transactional state, which suggests that bounded HCD should use the same overflow policy than LVM. This policy is based on aborting overflowed transactions and restarting them using LogTM-SE (BHCD) or BEVM (OBHCD), mixing eager and hybrid conflict detection. Some modifications on the conflict resolution policy have to be performed to preserve the isolation of transactions and to ensure that any hybrid transaction will finish after entering its commit phase. Notice that this is not a problem in LVM, because LVM guarantees that any transaction that reaches its end is coherent, and any requester will lose the conflict because the committing transaction has the ownership of the line.

HCD speculates with transactional stores, so a hybrid transaction only receives store requests that produce checks on the signatures when another hybrid transaction commits. Nonetheless, overflowed transactions execute eager conflict detection with requester-stalls policy, which might abort a hybrid transaction when it commits. This is problematic because transactions must be atomic, and stopping a hybrid transaction in the middle of its commit phase generates inconsistencies.

Hence, HCD needs to prioritize hybrid transactions that are in their commit phase. For that reason, any transaction that receives a conflict check request from a committing transaction must be automatically aborted. These transactions cannot be stalled because they are inconsistent.

Moreover, if a hybrid transaction receives a store request, it nacks the overflowed transaction, which will remain stalled until the hybrid transaction commits. In this occasion, eager transactions can be stalled because the conflicting location has been detected before it generates an inconsistence. However, previous accesses may not be isolated, and, in that case, the hybrid transaction will later abort the stalled transaction.

5.5 Bounded Lazy Conflict Detection

Previous Transactional Memory approaches solved overflows of the Transactional Buffer by executing overflowed transactions in LogTM-SE mode. However, merging transactions that perform eager version management with others that use lazy conflict detection might introduce incompatibility, which prohibits the overlap of both modes at the same time.

LCD doesn’t detect a conflict until a transaction reaches its end or conflicts with a committing transaction. As it performs lazy version management, transactions
always read values that are not modified by any transaction. However, LogTM-SE uses eager version management, so it speculatively modifies the memory hierarchy. If a transaction is aborted, a roll-back mechanism is activated to maintain the previous transactional state. Due to this, a lazy transaction can read locations modified by a LogTM-SE transaction without detecting any conflict, which breaks isolation. These eager transactional stores are not committed, and can be rolled back if the transaction aborts, which produces inconsistency in the lazy transaction.

This inconsistency cannot be detected at commit time because the Transactional Coherence does not care about reads, because it assumes that the memory hierarchy has non-transactional data. Even if the lazy transaction broadcasts its read set at commit time, the conflict may not be detected, because abort actions remove the transactional state of the eager writer.

An example of this inconsistency is shown in Figure 5.3. Processors 2 and 3 have overflowed their Transaction Buffers, so they abort their lazy transactions and restart them in LogTM-SE mode. Processor 2 reads in \( A \) the value of \( \text{size} \), let’s say that is \( k \), and performs some computation in the array, which causes the stall of transaction 3. After the loop, transaction 2 increments the size of the array in \( B \) and stores the result \( k+1 \) in the memory hierarchy. Then, transaction 1 reads \( \text{size} \) in \( C \), without detecting any conflict because it is executed in LCD mode. After that, it increments the size and updates the array location \( k+2 \). Unfortunately, transaction 2 aborts, restoring the old values from the log. When it restarts, it might read two different values on \( D \), and both of them will be wrong. If transaction 1 commits before the abort of transaction 2, \( D \) will read the old size \( k \), which is inconsistent because it supposes that transaction 1 hasn’t been executed. Otherwise, if transaction 1 is committed after the abort, \( D \) will read \( k+2 \), what means is wrong.

Figure 5.3: Conflict with LCD and LogTM-SE
5. Bounded Support for HTM Approaches

because transaction 2 doesn’t update the array. The right value should be $k+1$.

The impossibility to execute LCD and LogTM-SE transactions simultaneously suggests a conservative approach. As LogTM-SE is the mode used to handle overflows, LCD transactions have to switch to a safe mode compatible with eager version management. The candidates are EVM, LVM, HCD or even LogTM-SE. However, eager conflict detection only works if transactions are coherent from their beginning, a requirement that LCD doesn’t fulfill. Instead, HCD can manage inconsistencies by checking writes at commit time.

Our bounded LCD system executes lazy transactions until a transaction overflows, causing the abort and retrying the transaction in LogTM-SE (BLCD) or in BEVM (OBLCD). This overflow is notified to the other transactions, which switch from LCD to HCD. Of course, this switch doesn’t affect transactions at commit phase or waiting to commit. HCD mode is used when one transaction, at least, is run in LogTM-SE mode, so each processor has to maintain a counter with the number of transactions that have overflowed. When an overflowed transaction commits, this counter is decremented and, if no overflowed transactions remain, transactions switch again to LCD mode.

Figure 5.4 presents how BLCD solves the problem described in Figure 5.4. Transactions 2 and 3 overflow the Transactional Buffer, causing their re-execution in LogTM-SE mode and the immediate change of the other transactions to HCD mode. With this approach, transaction 1 will be able to detect the conflict before incrementing the size of the array, and it will stall its execution until transaction 2 releases its read-write set. Then, transaction 1 will update the $k+1$ location of the array.

Figure 5.4: Bounded LCD behavior
Chapter 6

Evaluation

This chapter presents a characterization of various HTM models with different benchmarks and workloads. Several statistics have been used to study the transactional behavior of each model and its advantages and disadvantages. Part of this chapter analyzes the main drawbacks of each model and attempts to assess if those problems can be solved.

6.1 Base System

We assume a CMP processor with a variable number of cores. It might contain from 1 to 32 cores, each configuration with its own interconnection network and memory controllers. Each core is an in-order, single-issue SPARC processor with 4-way 32 KB private L1 I&D caches and a 1MB shared L2 cache. Our CMP is a Non Uniform Cache Access (NUCA) system, where the L2 is distributed among the cores, as shown in 6.1.

Each configuration has its own interconnection network that uses 64-byte links and an adaptive routing. Each core is connected to its router, and the routers are connected creating a mesh. At most, four cores are connected to a memory controller to access the DRAM banks. 1 cycle is needed to connect a core with a router, 2 cycles to connect two routers and another cycle to perform the router computation. The memory controller needs 25 cycles to process a request.

A 32-core mesh has an expensive interconnection network, so an architectural decision has been made to simplify the 32-core CMP design. Each mesh node has two processors, each one with its L1 I&D, and a 1 MB shared L2 cache. Hence, the 16-core and 32-core configurations have the same interconnection network and L2 cache size. Other system features are described in Table 6.1.

6.2 Simulation Methods

The base system and the HTM models have been simulated using the Simics [28] simulation infrastructure from Virtutech and the Wisconsin GEMS toolset [31] to
6. Evaluation

Figure 6.1: Simulated CMP system

<table>
<thead>
<tr>
<th>System Model</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Cores</td>
<td>1,2 GHz in-order single issue</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32 KB 4-way, 64-byte line, writeback, 2-cycle latency</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1 MB 8-way per die, banked NUCA, writeback, 15 cycle-latency</td>
</tr>
<tr>
<td>Memory</td>
<td>4 GB, 4 banks, 300-cycles latency</td>
</tr>
<tr>
<td>L2 Directory</td>
<td>Bit vector of sharers, 6-cycle latency</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Mesh, 64-byte links 2-cycle wire latency, 1-cycle router latency</td>
</tr>
<tr>
<td>Transactional Buffer</td>
<td>Infinite (Base HTM)</td>
</tr>
<tr>
<td></td>
<td>32 entries (Bounded HTM)</td>
</tr>
<tr>
<td>Access-summary</td>
<td>Perfect Signatures (Base HTM)</td>
</tr>
<tr>
<td></td>
<td>2 Kb signature (Bounded HTM)</td>
</tr>
<tr>
<td>Commit Arbiter</td>
<td>10 cycles latency</td>
</tr>
<tr>
<td></td>
<td>(only HCD and LCD approaches)</td>
</tr>
</tbody>
</table>

Table 6.1: Base system parameters
build the memory environment. GEMS provided a LogTM-SE implementation. LogTM-SE has been slightly modified to better model the latency of the stores to the software log. Lazy Transactional Memory and other transactional support has been added to model the different HTM models described in the previous chapters.

6.3 Benchmarks and Workloads

We have selected a heterogeneous and representative set of transactional workloads from several benchmarks suites for our evaluation of the different HTM systems. The single-threaded LogTM characterization of the workloads is shown in 6.2. Barnes and Raytrace execute more transactions when the number of cores increases.

**SPLASH-2**

Splash-2 [55] presents a suite of benchmarks for multiprocessors, where lock-protected regions are turned into transactional blocks. As Splash-2 benchmarks have been optimized over the years to avoid synchronization, most of the time is spent in small, fine-grained transactions. This fact is not typical of Transactional Memory applications, who tend to have coarse-grain exclusive blocks. However, these workloads possess interesting properties that justify the analysis of their behavior in different HTM designs.

Barnes is a scientific application that implements the Barnes-Hut method to simulate the interaction of a system of bodies (N-body problem). Raytrace renders a three-dimensional scene onto a two-dimensional image plane using optimized ray tracing. Both workloads execute small transactions with high-contention properties. Although they represent a really small part of the code in sequential execution, when the number of threads increments, transactional execution becomes more important because more time is spent in resolving contention.

**Microbenchmarks**

Microbenchmarks are some benchmarks provided by GEMS 2.0 that interact with common data-structures. They present variable contention, depending on how data is distributed in these structures. These benchmarks regulate the contention by defining different parameters. The parameters have been defined to preserve the scalability in a reasonable manner.

Btree performs insertions and searches in a preloaded binary tree. The more insertions are performed in the tree, the higher contention the benchmark experiments. However, when the tree size increases, data is spread and contention is reduced. Btree executes coarse-grain transactions, inserting data in the tree half of the time. Deque, after some dummy work, inserts and deletes data from both ends of a queue using transactions. Concurrent inserts and deletes produce conflicts in the workload, so the amount of dummy work has been increased to reduce contention. This fact reduces the percentage of transactional cycles in the workload.

**STAMP**

49
6. Evaluation

<table>
<thead>
<tr>
<th>Workload</th>
<th>Bench. suite</th>
<th>Input</th>
<th>Num. Tx</th>
<th>Perc. Tx</th>
<th>Cycles Rx</th>
<th>RdLn Tx</th>
<th>WrLn Tx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>SPLASH-2</td>
<td>512 bodies</td>
<td>2187</td>
<td>2.9%</td>
<td>713.59</td>
<td>6.29</td>
<td>4.6</td>
</tr>
<tr>
<td>Btree</td>
<td>Micro.</td>
<td>50% inserts</td>
<td>100000</td>
<td>49.1%</td>
<td>733.06</td>
<td>13.3</td>
<td>1.72</td>
</tr>
<tr>
<td>Deque</td>
<td>Micro.</td>
<td>5000 junk work</td>
<td>100000</td>
<td>0.79%</td>
<td>62.05</td>
<td>2.5</td>
<td>2.89</td>
</tr>
<tr>
<td>Genome</td>
<td>STAMP</td>
<td>g128-s32-n65536</td>
<td>11003</td>
<td>99.49%</td>
<td>14121.5</td>
<td>47.61</td>
<td>8.66</td>
</tr>
<tr>
<td>Kmeans high</td>
<td>STAMP</td>
<td>m20-n20-t0.05 random50000-12</td>
<td>66667</td>
<td>6.44%</td>
<td>730.65</td>
<td>6.31</td>
<td>1.99</td>
</tr>
<tr>
<td>Kmeans low</td>
<td>STAMP</td>
<td>m40-n40-t0.05 random50000-12</td>
<td>66667</td>
<td>3.41%</td>
<td>734.97</td>
<td>6.31</td>
<td>1.99</td>
</tr>
<tr>
<td>Raytrace</td>
<td>SPLASH-2</td>
<td>teapot</td>
<td>47751</td>
<td>0.15%</td>
<td>8.26</td>
<td>6.26</td>
<td>4.63</td>
</tr>
<tr>
<td>Vacation high</td>
<td>STAMP</td>
<td>q60-u90 n32-r65536</td>
<td>4096</td>
<td>96.3%</td>
<td>40230.77</td>
<td>181.16</td>
<td>26.08</td>
</tr>
<tr>
<td>Vacation low</td>
<td>STAMP</td>
<td>q90-u98 n32-r65536</td>
<td>4096</td>
<td>98.5%</td>
<td>71213.46</td>
<td>298.43</td>
<td>26.84</td>
</tr>
</tbody>
</table>

Table 6.2: Transactional workloads properties
Characterization of the workloads when are executed in 1 thread and 1 core, what implies no contention.

STAMP [9] is the first transactional benchmark suite. The STAMP applications execute most of the time in transactions that deal with several software structures. As the programmer does not know how these structures are implemented, he has to ensure isolation when they are accessed. STAMP is a benchmark suite in progress, so we started to use STAMP 0.9.5 and later we updated to STAMP 0.9.7. These applications have been executed with the parameters that the last release proposes. However, some parameters have been increased (using the values proposed for non-simulated execution) to preserve the scalability in genome and vacation.

Genome implements a gene sequencing program that reconstructs the gene sequence from segments of a larger gene. Kmeans is a partition-based method and is arguably the most commonly used clustering technique in data mining. Vacation implements a travel reservation system powered by a non-distributed database. Client threads can introduce reservations to the database or can delete all reservations of a person, or search the database. Vacation low performs reservations most of the time, performing mostly queries and just a few modifications. Vacation high reduces the number of searches and introduces more operations that modify the database, causing more contention.

6.4 Experimentation

This section presents the characterization of transactional benchmarks, evaluating the performance and the scalability of the transactional memory models described in Chapter 4. The first section of this chapter compares transactions to locks and analyzes the benefits of using the Transactional Buffer instead of the software log, measuring which are the factors that determine the increment in performance. The
second section shows the behavior of the transactional models, arguing which are their advantages and disadvantages and exposing pathologies that damage their performance in some workloads. The last section of the chapter evaluates bounded hardware support. It evaluates the impact on the workloads when different Transaction Buffer sizes are used and discusses what the most reasonable Transactional Buffer size is.

6.4.1 LogTM characterization

LogTM is an optimistic HTM that works well when contention is low and the aborts are rare. However, recent studies [16] show its drawbacks when it has to deal with Transactional Memory workloads with huge transactions that increase contention. This section analyses its behavior and its weak points.

LogTM vs locks

Figure 6.2 and 6.3 compare the scalability of locks and unbounded HTM approaches. X-axis shows the number of cores, and the Y-axis the speed-up that they achieve respect sequential execution. In this Figure, we can see that lock-based programs serialize critical sections, arranging in sequence applications that spent most of the time in them, like Btree, Genome or Vacation. These critical sections modify linked structures that are updated at run-time, so the programmer must be conservative when they are accessed. Instead, LogTM executes these sections in optimistic transactions, and only serializes them when contention appears.

Locking behaves better with fine-grain critical sections, because it only serializes a small part of the program. However, an important overhead appears when several processors fight for the same lock, affecting the scalability of the application. Transactions don’t suffer from this pathology because they don’t require any synchronization when they start or finish, which allows LogTM to keep scaling in low-contention workloads like Kmeans.

LogTM analysis

LogTM performs well when contention is low, because commit actions are not required and aborts are infrequent. This happens when applications are run in configurations with few cores and small transactions, like in Barnes or Raytrace, or when contention is low, like in Kmeans. However, when contention is present, LogTM suffers from its software version management policy, wasting more time in recovery mechanisms and delaying the execution of critical transactions. It can be clearly seen in Deque, Genome or Vacation.

Figure 6.4 presents the normalized time of the different unbounded HTM approaches when are run in a 32-core environment respect their sequential execution. The graphic groups HTM approaches in benchmarks, and for each approach, presents how they spend their execution time. The time is distributed in four categories: non-transactional time, good transactional time (cycles spent in transactions that commit), transactional overhead time (non-useful transactional cycles, like stall,
Figure 6.2: Workload speed-up using different HTM approaches (a)

backoff or aborted work) and final barrier time (cycles wasted waiting the end of the final thread). The figure shows that LogTM is, with difference, the Transactional Memory approach with more transactional overhead.

In the majority of the workloads contention appears in configurations with many processors, given that more transactions are present at the same time. This
fact increases the probability of conflicts and puts more pressure on the interconnection network, which needs to send more transactional coherence messages. The transactional overhead associated with LogTM causes its bad scalability. Figure 6.2 shows that, for all the workloads except Kmeans, LogTM performs better on a 16-core configuration than on a 32-core.

Contention is not permanently present in transactional applications. Figure 6.5 categorizes transactions according to the contention that they experiment. Perfect transactions are those that don’t conflict, whereas stall transactions experiment conflicts, but can overcome them without aborting. 1 Abort are transactions that only abort once and 2+ aborts are transactions that need more than one abort to commit. The figure shows that, on average, 55.73% of LogTM transactions commit without experiencing any transactional overhead. However, 37.3% of the transactions abort more than once, which implies that transactions involved in conflicts tend to conflict again.

The big problem is that these conflicting transactions consume 93.05% of the transactional time and this hurts performance in transactional workloads. In Barnes, for example, 10% of the transactions consume more than 95% of the transactional
6. Evaluation

Figure 6.4: Normalized time distribution using 32-cores

time. Figure 6.6 summarizes how the transactional time among transactions is distributed according to Figure 6.5 categories. This fact suggests that LogTM optimistic policies may not be the best alternative to handle these conflicting transactions.

However, sometimes contention appears also with a few cores, usually in workloads with coarse-grain transactions. Some older studies [13] claimed that common-case transactions are short, but new transactional applications contradict this theory. Table 6.2 shows that Genome and Vacation execute transactions that access tens and hundreds of cache lines. These transactions need to store more lines in the software log, which enlarge transactional execution time, and spend more time recovering from an abort, given that more lines need to be restored. LogTM suffers from these overheads in Genome and Vacation, which limits their scalability and encourages other transactional techniques to reduce their execution time.

LogTM vs EVM

EVM uses the same Version Management and Conflict Detection policy as LogTM, but it replaces the software log with the hardware Transactional Buffer. This modification eliminates logging stores and reduces the time spent in the rollback mechanism. EVM outperforms LogTM in all the benchmarks by an average factor of 1.98x when run on an unlimited 32-core system, obtaining a maximum speed-up of 3.92x in Vacation-high.

Figure 6.4 shows that the difference in performance experimented in LogTM and EVM is caused by the overhead related with contention and the final synchro-
According to the figure, EVM and LogTM execute a similar number of good transactional cycles (time spent in transactions that commit), which suggests that logging is negligible. We implemented an EVM variant, called Elog, which stores in the software log all the modified lines, but, in case of abort, it recovers using the hardware mechanism. As it can be seen in Figure 6.2 and 6.3, the difference between EVM and Elog is minimal, corroborating the previous hypothesis. Elog behaves similar than EVM, that’s the reason why we decided to not include it.
6. Evaluation

<table>
<thead>
<tr>
<th>Workload</th>
<th>Stalls/ Aborts</th>
<th>Stalls/ Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LogTM</td>
<td>EVM</td>
</tr>
<tr>
<td>Deque</td>
<td>5.09139</td>
<td>10.18861</td>
</tr>
<tr>
<td>Genome</td>
<td>20.79</td>
<td>40.82</td>
</tr>
<tr>
<td>Kme-high</td>
<td>34.062</td>
<td>30.55</td>
</tr>
<tr>
<td>Kme-low</td>
<td>58826</td>
<td>4676</td>
</tr>
<tr>
<td>Raytrace</td>
<td>224198</td>
<td>133.522</td>
</tr>
<tr>
<td>Vac-high</td>
<td>63012</td>
<td>73719</td>
</tr>
<tr>
<td>Vac-low</td>
<td>60356</td>
<td>43116</td>
</tr>
<tr>
<td>Average</td>
<td>16.4492</td>
<td>180376</td>
</tr>
</tbody>
</table>

Table 6.3: Contention management in 32-core LogTM and EVM

in the other graphics.

Transactional overhead is the main drawback of LogTM. First, the software roll-back mechanism consumes 6.93% of the transactional time, whereas in EVM only consumes 0.06%. There are two reasons that explain this reduction. First, the Transaction Buffer allows hardware abort recovery, which accelerates the roll-back phase. Second, this fast-recovery mechanism reduces the number of aborts for a given conflicting transaction from 10 aborts to 6, removing an important part of the conflicts generated by aborted transactions that are still in their recovering phase. This fact reduces abort cycles and, what is more important, backoff cycles. As the backoff is exponential, the more aborts a transaction has, the more backoff cycles it executes. Table 6.3 shows the contention generated in EVM and LogTM.

Backoff cycles eliminate contention by separating conflicting transactions. That causes starvation and imbalance at the end of the workload, because conflicting transactions are delayed with respect to the others. Deque or Genome, for example, experiment more aborts in EVM than in LogTM, which suggests more contention and less performance. However, as the aborts are spread among transactions, the number of aborts per transaction is decreased, which reduces the backoff cycles and balances the work. Table 6.3 showsthat an aborting transaction in EVM only requires 15% of the LogTM backoff cycles before committing.

Some contention policies, like exponential backoff, increment the number of transactions that commit without being involved in any conflict because they spread contention. However, backoff may provoke imbalance, which is not desirable because the execution time is given by the execution time of the last thread. Figure 6.4 shows that EVM reduces a 42% the cycles spent in the last synchronization, and a 68% the transactional overhead respect LogTM, what suggest a correlation between contention management and imbalance computation.

6.4.2 Unbounded HTM approaches

Chapter 4 presented four HTM approaches that take advantage of an infinite Transactional Buffer to manage different Version Management and Conflict Detection
strategies. This section compares the base HTM approaches, analyzes their behavior in different applications and exposes the main drawbacks of each approach.

Figure 6.2 and 6.3 summarize the performance of HTM approaches in different workloads. Each workload has its own characteristics, providing a heterogeneous set of transactions with variable length and contention. Previous studies pointed minimal differences between HTM models. However, the following results present important divergence between approaches, depending on the workload, the contention and the system configuration.

**HTM behavior overview**

Each HTM offers a different alternative to detect and manage contention, and this fact determines how it will perform. Contention usually appears in multicore environments, where lots of transactions are executed at the same time and the interconnection network becomes a bottleneck. Mostly, differences between HTM approaches arise when configurations with 16 or 32 cores are used, but some differences can also be observed with fewer cores. Figure 6.7 shows which HTM approach obtains the best performance in each application and configuration. X-axis groups benchmarks, and for each benchmark, present a bar for configurations with different number of cores. The color of the bar corresponds to the best performer approach. Y-axis presents the speed-up achieved respect the sequential execution.

EVM performs better than other approaches with few cores because most of the workloads present low contention when few transactions are executed in parallel. EVM doesn’t require commit actions, which accelerates non-
conflicting transactions. Moreover, as aborts are rare, abort recovery and backoff cycles are negligible. Figure 6.2 and 6.3 show that, in an 8-core configuration, Btree runs in EVM 9% faster than in HCD or LCD, which serialize transactions at commit time to detect conflicts. However, when more cores are used, EVM generates more transactional overhead, which hurts its scalability. Instead, LCD serializes conflicting transactions, performing 35% better than EVM in Btree.

Nevertheless, the major differences appear in a 32-core environment, where each of the four approaches is the best performer on, at least, one application. LCD is the HTM approach that hoards more workloads, obtaining the best performance in Btree, Deque, Genome, and Raytrace. In Deque, LCD obtains a 2.23x speed-up with respect to EVM, because its Lazy Conflict Detection policy permits the separation of contention without using backoff. Except Genome, all of these workloads execute small and high contention transactions.

On the other side, EVM is the best performer in Vacation, no matter with high or low contention. That’s because Vacation execute huge transactions that conflict at the end of the transaction, when lots of useful work has been done. Eager Conflict Detection permits EVM to detect the conflict and to stall the conflicting transaction, restarting it again when the conflict disappears without re-executing the previous work. That’s the reason why EVM accelerates the execution of Vacation. Figure 6.2 and 6.3 show that EVM improves an 80% Vacation-low respect LCD.

All HTM approaches outperform LogTM for all workloads, because they show less transactional overhead. This fact suggests that some hardware support, like the Transactional Buffer, should be used in order to accelerate Version Management, because software recovering prevents scalability and generates contention.

EVM evaluation

In average, EVM performs 9% faster than LCD, 19% faster than HCD and 23% faster than LVM. However, Figure 6.2 and 6.3 show that there are several benchmarks where its transactional overhead is higher than HCD or LCD. In Raytrace, the transactional overhead associated with EVM is 30 times bigger than in LCD. In fact, if we discard the Vacation benchmark, EVM performs 26% and 24% worse than LCD and HCD, respectively.

Eager Conflict Detection allows EVM to detect conflicts when they are produced. To do this, loads and stores must be checked in order to guarantee consistency among transactions. This increases the time spent in transactions, which might produce more contention. In Barnes, the good transactional cycles are reduced in LCD by 73% with respect to EVM. Enlarging transactions usually introduces contention, causing more aborts.

EVM uses a requester-stall conflict resolution policy, aborting transactions that generate a deadlock cycle and performing a random exponential backoff before they are restarted. Stalling a transaction is a good technique in LogTM, because it might prevent software log recovery, or in large transactions, because it eliminates re-execution of useful work. However, stalls might not be desirable if the transaction
finally aborts, because they delay the restart and might produce new conflicts. In Genome, almost 10% of the transactional time is spent in stalled transactions waiting for the conflict time-out to expire, although just 11% of the stalled transactions don’t require to be aborted.

Figure 6.8 shows the distribution of the transactional cycles in percentage. First represent transactional cycles that haven’t been executed before, whereas retry are transactional cycles performed after an abort. Abort are cycles wasted in the rollback mechanism, stall are cycles performed until the end of a conflict and backoff are dummy cycles executed after an abort. Committing are cycles spent in moving new data to the memory hierarchy and waiting are cycles spent in the token queue. Notice that, on average, 34% of the transactional time is spent in the first execution of transactions, 26% in transactional cycles executed after an abort, 29% of the in backoff and 10% in stalls. Abort recovery requires only 0.1% of the total transactional time because the lines that need to be restored are usually present in the L1 cache.

EVM is not good at managing conflicts between small transactions with a high contention rate, where backoff is an important drawback. In Barnes, Btree, Deque and Raytrace more than 90% of the time is spent in transactions that abort, at least, twice (Figure 6.6). The reason is that more than half of the transactional time is spent in backoff and 30% in retry cycles. As small transactions do little work and are recovered rapidly, they might conflict several times, causing multiple aborts. The backoff is exponential, which means that the more times a transaction is aborted, the more overhead it generates. The inefficiency of managing huge contention hurts the scalability of EVM in Btree, Deque and Vacation, where the execution time in
a 16-core system is higher than in a 32-core system.

EVM is really efficient managing contention in workloads with large transactions. Its requester-stall policy allows transactions to reuse the transactional computation done before. In Vacation-high, EVM reduces by 50% the transactional overhead with respect to LCD (Figure 6.4), needing only the third part of the retry cycles. This fact permits a 32-core EVM to obtain a speed-up of 2x with respect to the LCD approach.

Surprisingly, EVM is the most balanced HTM approach (Figure 6.4). Although backoff delays the execution of conflicting transactions, Eager conflict detection and requester-stalls policy avoids the starvation of large transactions with long Read-Write sets.

**LVM evaluation**

LVM uses the same conflict detection and resolution policy as EVM, but it performs lazy instead of eager version management, replacing abort cycles with commit cycles. This is a drawback of our Transactional Buffer implementation, and other designs [47] that use the cache don’t need to perform a commit phase. LVM spends, on average, 1% of its transactional time moving transactional changes from the Transactional Buffer to the memory hierarchy (6.8). Notice that this is 10 times the cycles that EVM requires in abort recovery. However, some low-contention benchmarks like Kmeans-low spend 3% of the transactional time performing the commit and this percentage increases when the workloads are executed in environments with fewer cores.

Results of Figure 6.4 show that EVM performs 23% better than LVM. The reason why EVM more effective than LVM is that, more or less, a half of the transactions of the applications never abort, so EVM can avoid commit cycles on these transactions. Moreover, our experiments showed that most of the aborts are performed at the beginning of transactions, and, as the Transactional Buffer contains only few entries, the time spent in abort recovery is negligible. This is the case of Btree, where most of the conflicts involve read-only tasks that are resolved when the Transactional Buffer is empty.

The addition of commit cycles has another critical consequence for LVM. As all transactions are delayed, the number of conflicts increases in low-contention benchmarks, generating more stalls and backoff. Workloads with large transactions, like Vacation, also increment contention given that stalled transactions waste more time waiting for the committing transaction to finish.

On the other hand, LVM performs better than EVM in high contention workloads, like Deque, Kmeans-high or Raytrace, which requires few cycles to commit, although the number of aborts is high. As LVM doesn’t spent time in abort recovery, aborted transactions can be executed faster, removing possible conflicts. In these benchmarks, the number of transactions that are executed without getting any conflict in LVM is bigger than in EVM.

**LCD evaluation**
6. Evaluation

LCD performs Conflict Detection at the end of a transaction, which considerably increments the commit phase because each transactional write must check the access summary of other transactions. Furthermore, as transactional writes are buffered without any coherence request, the modified lines might not be close to the core, which delays even more the commit phase because they have to be brought to the L1 cache before being written. On average, LCD spends 7% of its transactional time in the commit phase (Figure ??), seven times more than LVM. However, in some high contention benchmarks like Deque, where the same cache lines are modified by several processors, commit phase spends up to 29% of the transactional time.

Lazy Conflict Detection uses a centralized arbiter to distribute the commit token, because Transactional Consistency only allows the commitment of one transaction at a time to preserve isolation of non-committed transactions. Due to this, when a transaction finishes, it has to ask the arbiter to the token, and if there is another transaction committing, it has to wait until the token is released.

We separate these waiting cycles in two classes. Waiting commit cycles represent the time that a transaction spent in the queue until it gets the token, whereas waiting abort is the time wasted in the queue before the abort generated by a previous transaction. Whilst waiting commit cycles delay the execution of transactions that don’t conflict, waiting abort cycles postpone the restart of conflicting transactions.

Figure (6.8) shows that LCD spends 6% of its transactional time waiting to commit and 3% waiting to abort. Even so, these undesirable cycles increase in applications where contention can be managed eager conflict detection. These applications concentrate lots of transactions that want to commit at the same time. As transactions are quite small and there isn’t non-transactional work to spread the computation, they waste most of the time in the queue waiting for the token. Barnes, for example, spends 39% of its transactional time waiting for the token before committing and 20% before aborting.

The strongest point of LCD is that it can eliminate backoff cycles from the transactional computation, which implies that transactions can be restarted just after the detection of a conflict. However, this fact increases the number of aborts, so the retry cycles are incremented. LCD performs twice the aborts with respect to EVM, spending 41% of its time retrying transactions. The first transactional cycles also increase because transactions do more work before they see the conflict.

LCD is really effective managing fine-grain workloads that mix non-transactional code with small transactions with a high contention rate. These applications serialize transactions, and, after some computation, contention can be effectively hidden by non-transactional execution. This technique reduces the number of conflicts, allowing the reduction of the aborts of Deque or Raytrace by more than an 80%. That’s the reason why these benchmarks experiment a 2.2x and a 1.6x speed-up when they are run in LCD instead of EVM.

An important drawback of LCD is the impossibility of stalling transactions
when a conflict occurs. As transactions speculate with their transactional accesses, each conflict implies the abort of the non-committing transaction because isolation is violated, and transactions must be restarted from their beginning, discarding previous work. This is quite drastic in applications with huge transactions, where a restart is expensive. This pathology can be seen in Figure 6.2 and 6.3, where Vacation doubles its transactional cycles respect EVM.

Another considerable problem of LCD is imbalance. When a transaction commits, it flushes its Transactional Buffer in its private L1 cache. If other transactions accessed this data, they must be restarted, generating a convoy. This pathology is reported in [7]. These transactions must get the data from a remote L1 when they restart. However, a processor that has recently committed a transaction usually finishes first if it executes another transaction that uses the same data, because data is placed closer to this processor (probably in its L1 cache). Hence, transactions that have committed recently tend to commit again, delaying the execution of older transactions. That’s the reason why LCD spends more than double the time in the last synchronization of threads with respect to EVM.

**HCD evaluation**

HCD mixes Conflict Detection policies to take advantage of the strong points of LCD and LVM. It performs Eager Conflict Detection in reads, stalling them to preserve isolation. Backoff is also needed to avoid repeated aborts. Backoff spreads the contention whereas stalls decrease the abort rate and prevent the re-execution of useful work. Although 38% of HCD’s transactional time is devoted to backoff, HCD performs similarly to LCD, because it reduces by 48% the abort rate and by 63% the retry cycles respect LCD (Figure 6.8).

Eager conflict detection also eliminates convoying by detecting read conflicts before commit time. These transactions can resolve the conflicts before, which eliminates the waiting to abort cycles and starvation. In Barnes, HCD only half of the waiting to abort and the synchronization cycles are needed (Figure 6.4), which allows the application to perform 18% and 39% faster than LCD and EVM. As we can see in Figure 6.2 and 6.3, HCD is the HTM that obtains the best balance in Barnes. However, there are other applications, like Btree or Vacation, where backoff delays the re-execution of transactions too much, causing the opposite effect.

The stall-requester policy is really effective to avoid the re-execution of transactional work. That’s the reason why EVM outperforms the rest of HTM approaches in Vacation, so it would be desirable to apply the same technique in HCD. However, HCD speculates with writes and, as these writes might be inconsistent, they have to be validated at commit time. If some transaction reads a value before the write is performed, the read becomes inconsistent and must be aborted when its conflicting transaction commits. HCD cannot stall writes, which causes poor performance in Vacation.
6.4.3 HTM approaches with bounded support

The previous section evaluates the base HTM approaches with an unlimited Transactional Buffer and perfect signatures. This section evaluates the performance of the different HTM approaches with a bounded Transactional Buffer and finite signatures.

Signatures

All the unbounded HTM approaches use signatures to track the transactional accesses. However, small signatures might introduce false conflicts, which increases the number of aborts and the transactional overhead. Table 6.4 shows the percentage of conflicts introduced by both read and write signatures in a 32-core EVM environment. Bounded HTM approaches use 2048-bit signatures, which doesn’t affect the performance of most workloads. Nevertheless, large read-only transactions, like Vacation-low, require bigger signatures.

<table>
<thead>
<tr>
<th>SIGNATURE SIZE</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE WORKLOAD</td>
<td>R (%)</td>
<td>W (%)</td>
<td>R (%)</td>
<td>W (%)</td>
<td>R (%)</td>
</tr>
<tr>
<td>barnes</td>
<td>0.08</td>
<td>0.05</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>btree</td>
<td>0.01</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>deque</td>
<td>0.0</td>
<td>0.01</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>genome</td>
<td>49.35</td>
<td>0.31</td>
<td>4.93</td>
<td>0.02</td>
<td>0.33</td>
</tr>
<tr>
<td>kmeans-high</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>kmeans-low</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>raytrace</td>
<td>37.66</td>
<td>0.0</td>
<td>15.24</td>
<td>0.0</td>
<td>1.48</td>
</tr>
<tr>
<td>vacation-high</td>
<td>12.17</td>
<td>0.81</td>
<td>10.73</td>
<td>0.03</td>
<td>3.83</td>
</tr>
<tr>
<td>vacation-low</td>
<td>95.81</td>
<td>1.11</td>
<td>92.44</td>
<td>0.11</td>
<td>82.59</td>
</tr>
</tbody>
</table>

Table 6.4: False positive rate using signatures

Transactional Buffer

HTM approaches used an infinite Transactional Buffer to improve Version Management. However, this is an expensive hardware structure, and it should have only a few entries. After running the applications with different HTM approaches and classifying transactions by their length, we found that a 32-entry Transactional Buffer handles most of the transactions. Table 6.5 shows the percentage of committed transactions that fit in the Transactional Buffer using different sizes.

Performance

Bounded HTM approaches have been evaluated using different configurations and HTM approaches. We would like to note that Deque, Kmeans and Raytrace execute small transactions, so none of them will overflow the Transactional Buffer. Simulations showed that the percentage of false positives in these benchmarks is negligible, performing as their unbounded HTM approaches. We decided to not evaluate them in this section because the behavior of these benchmarks is not degraded when bounded resources are used. On the other hand, Vacation always
overflows the Transactional Buffer, so a bigger structure should be used to handle most transactions by hardware.

Figure 6.9 shows the scalability of the rest of the workloads when a 32-entry Transactional Buffer and 2048-bit signatures are used. We can see a considerable degradation in performance in comparison to Figure 6.2 and 6.3, even when it affects just a small set of transactions. BLCD augments three times its execution time in Genome with respect to LCD, whereas BEVM runs Vacation-high four times slower than EVM.

BLVM, BHCD and BLCD perform similar to LogTM-SE in the majority of the workloads. That’s because overflowed transactions are the ones that generate more contention, and, since they account for most of the execution time, the end behavior is similar than LogTM-SE. Only in Barnes we can see a difference between BLCD or BHCD and LogTM-SE, but BHCD degrades 27% its performance with respect to HCD. Moreover, as an overflow causes the abort of a given transaction, all its previous useful work has to be discarded. That’s the reason why a Vacation-low 1-thread execution performs 31% worse in these modes than in LogTM-SE.

However, BEVM improves LogTM-SE performance because it only aborts by software when the Transactional Buffer is exceeded. This is really effective in a requester-stall conflict resolution policy, where small transactions usually are aborted by larger transactions that have accessed this data before. This fact reduces the number of software aborts in Btree and Genome, obtaining similar behavior to EVM. Furthermore, overflows don’t discard previous work, which accelerates the commit of an overflowed transaction. BEVM only loses 1% with respect to EVM in Genome because most of the aborted transactions can be recovered by hardware.

The good results obtained by BEVM encouraged its usage in bounded versions of LVM, HCD and LCD approaches. In their optimized design (OBLVM, OB-HCD and OBLCBD), overflowed transactions are executed in BEVM mode instead of LogTM-SE mode. Thus, software recovery is only performed when an aborted transactions exceeds its resources. Figure 6.10 shows the speed-up achieved by optimized

| TRANS. BUF. SIZE/ | 16   | 32   | 64   | 128  | 256  |
| WORKLOAD        |      |      |      |      |      |
| barnes          | 83.19| **94.79** | 94.79 | 95.72 | 100  |
| btree           | 77.17| **90.59** | 97.18 | 100  | 100  |
| deque           | 100  | 100  | 100  | 100  | 100  |
| genome          | 3.25 | **99.14** | 99.57 | 99.69 | 100  |
| kneans-high     | 100  | 100  | 100  | 100  | 100  |
| kneans-low      | 100  | 100  | 100  | 100  | 100  |
| raytrace        | 100  | 100  | 100  | 100  | 100  |
| vacation-high   | 0    | **0.39** | 5.66  | 92.38 | 99.9 |
| vacation-low    | 0    | 0    | 0.88  | 90.43 | 99.9 |
Figure 6.9: Performance of Bounded HTM approaches
6. Evaluation

versions of bounded HTM approaches.

Optimized approaches improve the performance of BLVM, BHCD and BLCD in Barnes, Btree and Genome. In Barnes, 10% of transactions overflowed resources in BHCD, causing the 56% of the aborts of the whole execution. However, only a 12% of the aborts produced by overflowed transactions exceed resources, so the majority of aborts are handled by hardware. That’s the reason why OBHCD improves an 11% BHCD execution in Barnes. More spectacular is the improvement in 32-core Genome, where OBLCD gets a 2.64x speed-up respect BLCD. However, Btree cannot take much advantage of lazy policies because overflowed transactions are the ones that generate contention and they are executed eagerly.

Nevertheless, Vacation shows that sometimes all transactions overflow the buffer, and, as it is a high contention workload, old transactional state has to be recovered several times using the software. All transactions are executed in LogTM-SE mode, so no bounded HTM approach is able use its strong points in contention management. Hence, new techniques should be proposed to accelerate overflowed transactions by hardware.

Signature length can also be a problem in large-transaction applications where the false-positive rate might be high. Vacation-low, for example, doubles its execution time in an 8-core 2Kb-signature LogTM-SE system respect to a LogTM system that use infinite signatures. Therefore, new mechanisms should address the problem of false-positives of signatures in large transactional applications.

Summarizing, LogTM-SE offers an alternative to handle overflows in bounded HTM approaches. Hybrid Transactional Memories use the same technique using a STM when an overflow occurs. Although previous studies said that HyTMs are good at managing overflows, this study shows that overflowed transactions are the main bottleneck in transactional applications, and they should be accelerated by hardware. Moreover, transactional applications tend to use coarse-grain transactions, which will damage even more their performance when little hardware support is provided.
6. Evaluation

Figure 6.10: Performance of Optimized Bounded HTM approaches
Chapter 7

Conclusions

This Master Thesis has presented several HTM approaches that use different techniques to manage contention. LogTM employs a software log to perform version management, whereas other approaches use the hardware Transactional Buffer, which can be used eagerly or lazily, to accelerate aborts or commits.

LogTM performs better than Locks, because non-blocking synchronization allows the execution of transactions that don’t conflict with the rest. However, LogTM isn’t good managing contention because it recovers from aborts by software, which causes a performance degradation in workloads with large transactions or in many-core environments.

Other HTM approaches have been modeled and their effectiveness has been proved in some transactional workloads. Their evaluation has shown that each one has its pros and cons, and their behavior depends on the workload and the configuration. Although the differences in a given application are considerable, none of the HTM approaches outperform the rest in average. This fact suggests that a HTM must offer different execution modes to use the best approach for each application.

EVM accelerates LogTM’s roll-back mechanism by storing old values in a Transactional Buffer, what removes explicit logging and reduces abort recovering and contention. EVM doubles the performance of LogTM, but it still creates lot of transactional overhead in high-contention applications that prevents its scalability. EVM performs eager conflict detection and version management, what allow transactions to stall their execution to preserve isolation. This is good for applications with some contention and large transactions, because it prevents aborts. LVM is identical than EVM, but it uses Lazy Version Management. Although it reduces the execution time in high-contention workloads with small transactions, some commitment actions have to be done, what degrades the performance in workloads with low contention and large transactions.

LCD uses Lazy Conflict Detection, delaying conflicts until commit time. This technique is useful in high-contention workloads with small transactions, because it reduces the transactional overhead generated by stalls and backoffs. However, it performs poorly when contention is present in applications with large transactions,
because each conflict aborts non-committed transactions. LCD also increases the unbalance of the workloads.

We presented another HTM approach, HCD, that tries to mitigate the pathologies of LCD by detecting read conflicts eagerly. This fact reduces the number of aborts, but backoff might increase unbalance and transactional overhead. Although HCD and LCD perform similarly, HCD improves LCD because it reduces the number of requests that the commit arbiter has to deal with. However, it performs worse than EVM managing large transactions.

We have seen that an appropriate HTM approach provide important benefits in the execution of transactional applications. However, transactional overhead and unbalance have noteworthy weight in high-contention applications, especially in those ones with large transactions. Other approaches with different transactional mechanisms and new contention policies may improve the current techniques.

This Master Thesis has also proposed how previous approaches can use bounded hardware support. Transactional Buffer overflows are handled using LogTM-SE mechanisms. Hence, our system supports the execution of different transactional modes at the same time. Each mode represents a HTM approach, and this fact can produce some inconsistencies among transactions that execute different modes. We have presented some techniques to address these problems.

We have evaluated different HTM approaches with finite hardware support. Signatures are useful in the majority of the workloads, but few of them with huge transactions experiment degradation in performance due to false positives. Some applications with small transactions never overflow the Transactional Buffer, whereas others commonly do it. BEVM is the only HTM approach that can efficiently handle the overflows because it just executes software mechanisms when it is necessary, whereas other bounded approaches abort overflowed transactions and re-execute them in LogTM-SE mode. These approaches perform similarly than LogTM-SE because overflowed transactions generate unbalance and contention. Optimized bounded HTMs use BEVM to handle overflows, what improves the performance of finite lazy approaches.

Transactional Buffer is an expensive hardware structure that should be small to be realistic. Proposed solutions to manage overflows are not good enough and other alternatives should be conceived in order to take advantage of each HTM approach when overflows are common. New proposals that involve cache buffering or coherence protocols might increase the efficiency of lazy approaches, and, at the same time, reduce the hardware that they require.
Appendix A

A transactional coherency protocol

This appendix presents the transactional coherency protocol proposed in LogTMSE, called MESI CMP filter directory, designed for a two level cache hierarchy. A private first level of cache ($L1$) keeps the data close to the processor, and an exclusive shared second level of cache ($L2$) records the owner of a line in a directory. This protocol keeps the coherency in non-transactional code and, at the same time, detects conflicts among transactions, guaranteeing isolation and consistency. This is the protocol that has been used in the different implementations modeled in the previous chapters.

This protocol uses a typical MESI directory in non-transactional code. A line in the first level of cache ($L1$) can be found in one of its four states (Modified, Exclusive, Shared or Invalid), and a $L2$ directory keeps the lines that are held in the $L1$. The directory tracks the owner of a modified line or the list of sharers in case of multiple readers. For that reason, all transitions, except from exclusive to modified, must inform the $L2$ due to keep the consistency in the directory.

When a line is requested in a processor, it sends a message to the $L2$ directory, who informs the owners of the line in case of a read, or the owner in case of a write. When the remote processors receive the message, it sends the data to the requester, who shares the lines with the owner. If no processor owns the line, the $L2$ serves it.

The $L1$ transactional coherency protocol, what includes states, transitions and communication with the $L2$, doesn’t change very much from a typical MESI. There are only two differences. The first one, when a processor receives a read (Fwd-GETS) or a write (Fwd-GETX) from the directory, it checks its write or read-write signatures. If the requested address is in a signature, the request is replaced with a Fwd-GETX-S or Fwd-GETX-X. In that case, the line has been used in a transaction, so the processor has to invalidate the requester with a NACK signal. When the requester collects all the responses from other processors, it receives a NACK-final if there is a conflict. Otherwise, it acts like a classical MESI protocol.

The second difference resides on the transactional evicted lines. In a typical MESI, and evicted line provokes a data update on the $L2$, an actualization in the directory and a transition in the $L1$ from M or E to I. However, we need
Figure A.1: L1-cache transactional MESI coherency protocol
A transactional coherency protocol

A mechanism to detect those transactional lines that are evicted from the cache to keep the coherency among transactions. First, the system must identify those evicted lines that are used in an active transaction. Basically, the system checks if the line is in the processor’s signatures, and, in that case, sends a special message to the directory (PUTX-X), who puts the line in a special state, called sticky state.

The $L2$ cache protocol manages the directory and keeps the transactional state coherent by checking the signatures of the processors that own the line. The $L2$ cache is shared among all processors and every line must be in one of the following state. NP defines when a line is not present in the $L2$, SS is a line shared among more than one processor, MT if the line is modified in only one processor (what implies that $L2$ does not have the right value) or processor has that line exclusively, and M, and sticky state for those transactional lines that have been evicted from $L1$ cache.

The first time that a line is brought from memory to the $L2$ (initially, in the NP state), the system needs to broadcast a checking-signature request to ensure that nobody has used that line before in a transaction. All the processors check their signatures and send an ACK or a NACK to the requester. When it collects all the responses, detects a conflict if it is at least one NACK (what implies to stall or abort), or receives the line from the $L2$ if all the responses are ACKs.

When a line is shared (SS) in the $L2$ cache, it means that more than one processor has a copy of the line, and the line is not modified. On a read request (GETS), the $L2$ provides the line to the requester. However, a write request (GETX) needs to invalidate all the copies in other processors, what implies an INV request to the members of the sharer list of the directory. When this message arrives to a processor, it checks its signatures, and, if the line has been used in the current transaction, it invalidates the requester using a NACK.

Similarly, a line can be in the $L2$ in the MT state. This means that the line is modified in a private cache. Any request to that line is forwarded to that processor, who will check its signatures to detect conflicts and NACK the requester in that case. Otherwise, it sends the line to the requester and the $L2$, and updates the $L2$ directory, which will mark that line as a shared.

Finally, a line is marked as an M when it has been modified but it is not present in any private cache. This means that a modified line has been evicted from the $L1$ cache. It also could be that a transactional evicted from the $L2$ arrives from memory, even if it was a read line. If this is a transactional line, M state works as a sticky state, who keeps who has read or modified that line. When a memory request arrives, the directory forces a selective check in the signatures in the processors that owns the line.

Notice that a line leaves the sticky state when a transaction commits, because then the signatures are cleaned and future checks will not cause conflicts.
A transactional coherency protocol

Figure A.2: L2-cache transactional MESI coherency protocol
The transactional coherency protocol can also be adapted to a Lazy Transactional Memory environment. First, transactional lines are never kept in the sticky state. This means that evictions from cache are classified as a non-transactional code, what eliminates signature’s checking in the M state. The only exception is found when a transaction is committing, because that line can not be accessed until the commit is completed, due to preserve atomicity.

The second change is removing some filter checks. For example, transitions from NP do not need to broadcast signature checking, as transactions detect conflicts lazily. However, we decided to not change the protocol to preserve a unique protocol for all the HTM implementations presented. Instead, in lazy environments signatures are ignored in case of conflict in the middle of the transaction, emulating a lazy conflict resolution.
Bibliography


A transactional coherency protocol


A transactional coherency protocol


77
A transactional coherency protocol


A transactional coherency protocol


A transactional coherency protocol
