Parallel-Architecture Simulator Development Using Hardware Transactional Memory

Adrià Armejach

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

A thesis submitted for the degree of

Master in Information Technology

September 2009

Director: Adrián Cristal
Co-Director: Osman Unsal
Tutor: Eduard Ayguadé

Acknowledgements

Finishing this master thesis ends one chapter of my life and marks the beginning of another.

I would like first to specially thank Marina, for supporting me through the whole process and encourage me to keep learning.

I, of course, would not be here today without the help of my closest people, thanks for being there. My father gets some extra thanks for reading and editing this document.

I must thank all my collages, friends I should say, that made this a wonderful experience. I felt part of the group since the very first day, and they helped me with everything they could. Specially, I would like to thank Saša for being my I-need-help guy, without his help I could not have gotten through; and Ferad for his reviews and nonsense funny laughs.

I also thank my supervisors and tutor, for bringing me this opportunity and helping me with their experience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis has been developed in the joint center held by Barcelona Supercomputing Center (BSC) in collaboration with Microsoft Research (MSR). The Center focuses on the design of future microprocessors and software for the mobile and desktop market segments basis (1).

The following sections of this chapter discuss the context and motivations of this thesis, along with its scope, main objectives and contributions.

1.1 Context

During the last decades the number of transistors in a single chip has increased exponentially, from the first home computers that had a few thousands of transistors (∼6.000) to today's designs that involve hundreds of millions (∼700.000.000) (6). Given this ever-increasing transistor densities (see Figure 1.1), and mainly in response to the problems of incrementing performance in single-threaded processors that computer architects faced over the last years (e.g., undesirable levels of power consumption because of high clock frequencies (15)), manufacturers have shifted to multiprocessor designs instead. Large-scale multiprocessors with more than 16 cores on a single board or even on a single chip will soon be available. This is achieved by using a simpler and smaller core processor design and replicating it many times, reducing power requirements and making thread-level parallelism the new challenge to achieve high performance.

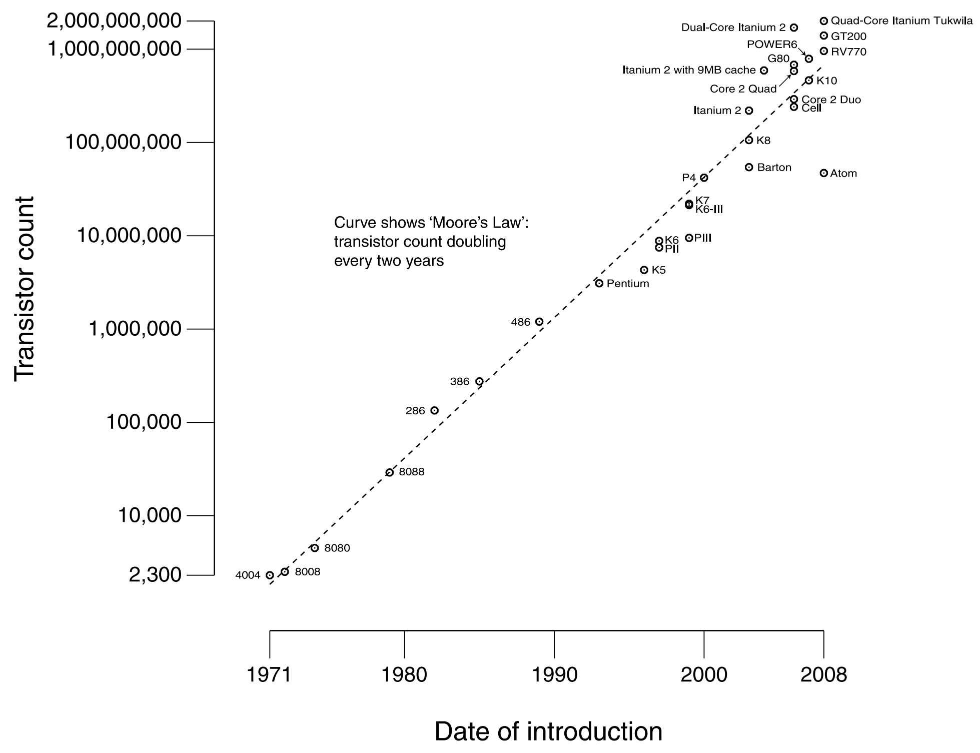


Figure 1.1: Curve showing CPU transistors count 1971-2008.

The advent of chip multiprocessors (CMPs) has moved parallel programming from the domain of high performance computing to the mainstream. Now, software developers have the difficult task to write parallel programs to take advantage of multiprocessor hardware architectures.

1.1.1 Why is Parallel Programming Difficult?

Parallel programs are more difficult to write than sequential ones, concurrency introduces several new classes of potential software bugs, of which race conditions (e.g., data dependencies) are the most common ones (25). To take advantage of thread-level parallelism involves creating several parallel tasks that need to synchronize and communicate to each other. Today's programming models commonly achieve this via lock-based approaches, in this parallel programming technique, locks are used to provide mutual exclusion for shared memory accesses that are used for communication among parallel tasks.

Unfortunately, when using locks, programmers must pick between two undesirable choices:

- Use coarse-grain locks, where large regions of code are indicated as critical regions. This makes the task of adding coarse-grain locks to a program quite straightforward, but introduces unnecessary serialization that degrades system performance.
- On the other side, fine-grain locks cause critical sections of minimum size. Smaller critical sections permit greater concurrency, and thus scalability. Anyway, this schema leads to higher complexity, and makes it more likely to have problems such as deadlock.

There are many other problems related with locks, some of the most important are: priority inversion, convoying and debugging. The first one prevents high priority threads to be executed if a low priority one is holding the common lock, while in convoying all the other threads of the system have to wait if the thread holding the lock gets de-scheduled due to an interrupt or a page fault; finally, lock-based programs are difficult to debug, since bugs are very hard to reproduce themselves (8).

1.1.2 Transactional Memory

To address the need for a simpler parallel programming model, Transactional Memory (TM) has been developed and promises good parallel performance with easy-to-write parallel code (22; 25).

Unlike lock-based approaches, with TM, programmers do not need to explicitly specify and manage the synchronization among threads; however, programmers simply mark code segments as transactions that should execute *atomically* and *in isolation* with respect to other code, and the TM system manages the concurrency control for them. It is easier for programmers to reason about the execution of a transactional program since the transactions are executed in logical sequential order according to a serializable schedule model.

TM allows non-blocking synchronization with coarse-grained code, deadlock and livelock freedom guarantees. Non-blocking synchronization is achieved by executing the transactions optimistically (in parallel), while still guaranteeing exactly-once execution as if these were ran in isolation and committed in a serial order. If any conflicts are detected during the execution, some of these transactions will be aborted to maintain system consistency.

TM can be implemented either in software (STM) or hardware (HTM). STMs are more flexible but suffer from serious performance overheads whereas HTMs are faster but limited due to hardware space constrains, even though this space limitations can be handled using virtualization or other mechanisms (30).

As more processing elements become available, programmers should be able to use the same programming model for setups of varying scales. Hence, TM is of long-term interest if it scales to large-scale multiprocessors properly.

1.2 Project Objectives

In the previous section, we stated several problems regarding the actual programming models when developing parallel applications, and how new approaches like TM are coming up to solve those problems. The main motivation for researching new ways to face parallel programming issues is that manufacturers are turning to CMPs in detriment of single-threaded processors as a realistic path towards scalable performance for server, embedded, desktop and even notebook platforms (16; 26).

This project aims two main goals. The first one is to develop a Hardware Transactional Memory (HTM) module based on an already existing protocol, and integrate it into a full-system simulator. Evaluate its performance using realistic benchmarking tools and extract conclusions about its scalability and performance.

The second one is to provide a tool to experience with its many configurable parameters, to see how different setups of the components (e.g., size or associativity of the caches) affect the performance of this kind of systems and to be able to detect its associated pathologies or if any bottleneck exist. Furthermore, it can also be used to compare new approaches that are under development (35) or that can be implemented in a near future in the Center against an existing state-of-the-art proposal.

1.3 Project Description

The protocol chosen to implement the HTM module is the first non-blocking TM implementation that targets distributed shared memory (DSM) systems, and uses directory-based mechanisms to ensure coherence and consistency. By targeting DSM-like systems we are focusing in the domain of high performance computing sector. This protocol is called Scalable Transactional Coherence and Consistency (Scalable-TCC) (19).

With regard to the simulator, we used The M5 Simulator: a modular platform for computer system architecture research, with system-level architecture as well as processor micro-architecture basis (12). It supports multiple Instruction Set Architectures (ISAs) such as Alpha, SPARC, MIPS, and ARM, with X86 support in progress. However, full-system capability is only available in Alpha and SPARC. For our project we will use Alpha architecture since it is the only one that can boot Linux 2.4/2.6, and we are more familiar with this OS than others.

In order to achieve our objectives the whole project workload has been split in three work packages:

- Introduction and generic modifications to The M5 Simulator.
- Scalable-TCC HTM module.
- Merge Scalable-TCC HTM module into The M5 Simulator.

In the **first package**, once the system was set and running, we learned about The M5 Simulator structure and its tools. Moreover, some generic modifications have been done to the simulator that will be useful in the future when merging the Scalable-TCC HTM module into it (e.g., extend the ISA to support instruction to begin and commit a transaction).

The **second package** has the largest volume of work. First of all, a considerable amount of time was spent on reading about TM and directory-based protocols, to have a general knowledge to be able to analyze properly the description proposed on the Scalable-TCC paper. Afterwards, before starting to specify and implement the HTM module, some decisions about the cache hierarchy setup and how to implement DSM were made. The implementation of the HTM module was split into two different and isolated sub-projects:

- A cache-simulator with directory that fits to the Scalable-TCC directory-based protocol demands. For this module an already existing cache-simulator for MESI protocol was at our disposal. Nevertheless, Scalable-TCC protocol has been proven to behave quite different than common MESI/MOESI like protocols, besides the fact that it was not prepared for DSM-like systems nor for a directory-based protocol. Therefore, all the functionalities have been written from scratch.
  This module allows the creation of a cache hierarchy (L1, L2, etc.), it also handles main memory using DSM schema and the directory structures. Statistics are incorporated at cache level (i.e., for every cache level), at main memory, and at directory level. Tracking events such as: hits, misses and ticks spent, amongst others; that occur on these mentioned structures.

- The HTM module will interact with the previous one to define the desired memory hierarchy for each node of the system. The module defines the concept of transaction and stores the necessary information such as the transaction state and the transaction identifier. The interface of the HTM is also defined here, so here are defined the behaviors, for example, of the *begin* and *commit* transaction instructions, among others.
  Since we want a powerful statistics system, there will be statistics at transaction and at HTM level too; regarding to the number of memory accesses, *begin* instructions executed, etc.

By splitting the work in two smaller sub-projects we can trace errors easier than having a larger single project. For this reason, each subproject has been checked separately using unit tests to verify its functionalities, and using some stress tests afterwards, to ensure a higher degree of correctness before starting the last work package.

In the **third package** of this project we merged the HTM module into The M5 Simulator. During this process some modifications on both sides, the HTM module and the simulator were really necessary. For example, in the simulator side, we had to declare the memory hierarchy and its parameters using our implementation. Some more complex modifications regarding fault tolerance and switching from user-mode to kernel-mode (and vice versa) were also necessary.

After checking that the integration was working as expected, we started with the testing phase using a couple of basic tests (e.g., incrementing a shared counter using transactions). Once those test were passed correctly, a set of synthetic benchmarks that are intended to represent real applications that cover a wide range of transactional scenarios (17), were used to evaluate performance and scalability of the system.

1.4 Contributions

Basically, this thesis presents the implementation and evaluation of a hardware transactional memory system. Its major contributions are:

- Propose an implementation of a scalable design for TM that is non-blocking and is tuned for continuous transactional execution.
- Provide a memory system, that allows for a configurable number of cache entries, associativity, cache-line size, and all the access timings in the memory hierarchy.
- Provide a powerful statistics system to extract conclusions about the transactional executions. In particular, we have stats for every level of cache, main memory bank, directory, transaction and at global HTM level.
- Demonstrate that the proposed TM implementation scales efficiently to 32 processors in a DSM system for a wide range of transactional scenarios.

1.5 Organization

After introducing the context, and state the objectives and contributions of the project, in this section, we explain the structure of the document with a brief description of each chapter.

The second chapter introduces basic concepts about transactional memory, discussing different possible design approaches, stating its advantages and disadvantages.

The third chapter explains in detail the protocol that our HTM simulator will use and the configuration choices done at this moment in the simulator side.

The fourth chapter presents the related work, where we talk about some of the most relevant proposals regarding transactional memory.

The fifth chapter introduces the architecture details that our proposal will use, at processor, caches, directory and interconnection network level.

The sixth chapter presents the cache with directory module, with a general detailed explanation, the functionalities and configuration possibilities it implements, the available statistics and a unit test example showing its execution statistics output.

The seventh chapter follows the same layout than the previous one, but explaining the HTM module.

The eighth chapter explains the merging process of the HTM module and The M5 simulator, along with the modifications required in the simulator side and the problems encountered.

The ninth is the evaluation chapter, where we introduce the system configuration used to run the tests, the methodology, and finally we present and discuss our results.

The tenth chapter explains the initial planning and its modifications, the project costs, achieved objectives, exposes future lines of work and personal conclusions.

Chapter 2

Transactional Memory

In this chapter we introduce some basic concepts about Transactional Memory (TM), like its properties, along with the mechanisms and policies used to guarantee those properties. We also discuss the advantages and disadvantages of Hardware and Software based systems.

2.1 Transactional Memory Basics

Using conventional multi-threaded programming on shared memory machines, programmers need to manually manage the synchronization among threads when they use locks. For example, they must select the lock granularity, create an association between shared data and locks, and manage lock contention. In other words, with locks, programmers not only need to declare where synchronization is used, they must also implement how synchronization will occur.

In contrast, with Transactional Memory (TM), programmers simply declare where synchronization occurs, and the TM system will handle the implementation properly. In more detail, with TM, programmers indicate that a code segment should be executed as a transaction by placing that group of instructions inside an *atomic block* (28) as shown in Figure 2.1.

It is the responsibility of the TM system to guarantee that the transactions have the following properties: atomicity, isolation, and serializability. Firstly, *atomicity* means that, either all or none the instructions inside a transaction

```
atomic {
    if ( foo != NULL ) a.bar();
    b++;
}
```

Figure 2.1: Group of instructions representing a *transaction*.

must be executed. Secondly, having *isolation* means that none of the intermediate state of a transaction is visible outside of the transaction (i.e., any memory update is visible to other threads during the execution of a transaction). Finally, *serializability* requires that the execution order of concurrent transactions is equivalent to some sequential execution order of the same transactions (24).

The way that TM systems achieve good parallel performance is by providing optimistic concurrency control (19), where a transaction runs without acquiring locks, optimistically assuming that no other transaction operates concurrently on the same data. When the TM system executes the body of an *atomic block*, it is done speculatively (hence the name optimistic). While the body is executed, any memory addresses that are read are added to a *read set*, and the ones that are written are added to a *write set*. Finally, at the end of the atomic block, the TM system ends, or *commits*, the transaction.

To verify that the speculative execution of the transaction is valid, the TM system compares the read and write sets of all concurrent transactions. This allows to perform fine-grain read-write and write-write conflict detection (i.e., to know the exact conflicting address). If no conflicts are detected, the transaction commits successfully, otherwise it is aborted, and the execution is *rolled back* to the beginning of the atomic block and *retried*.

The key idea of TM is that because of their atomicity, isolation, and serializability properties, transactions can be used to build parallel programs. Using large atomic blocks simplifies parallel programming because it provides ease-of-use and good performance. First, like coarse-grain locks, it is relatively easy to reason about the correctness of transactions. Second, to achieve a performance comparable to that of fine-grain locks, the programmer does not have to do any extra work because the TM system will handle that task automatically for him (24).

2.2 Transactional Memory System Taxonomy

Like in database systems, there are a variety of ways to provide transactional properties of atomicity and isolation. There are three categories that can be used to classify TM systems: the *data versioning* mechanism, the *conflict detection* policy, and the approach used for implementation, being *hardware-based* or *software-based* (24; 27).

2.2.1 Eager versus Lazy Data Versioning

Transactional systems must be able, at least, to deal with two versions of the same logical data. An updated version and an old version, just in case the transaction fails to commit, so the old version can be used to roll back. Updates to memory addresses, when a write is executed by a transaction can be handled either eagerly or lazily (27).

In *lazy data versioning*, updates to memory are done at commit time. New values are saved in a per transaction store buffer, while old values remain in place. This grants *isolation*, because all the speculative updates are not visible by other threads until the transaction commits. On the other side, *eager data versioning* applies memory changes immediately and the old values are stored in an undo log. If the transaction aborts, the undo log is used to restore memory changes. Note that in order to grant *isolation* in eager TM systems, transactionally modified variables must be locked, and therefore they cannot be accessed until the owner either commits or aborts the transaction. This can derive into a classic deadlock situation, as shown in Figure 2.2, where transaction T_1 has acquired the lock for memory address `A` and is waiting to acquire the lock of memory address `B`, and T_2 has acquired the lock for address `B` and is waiting to acquire `A`'s address lock. Therefore, a *contention management* mechanism is required, and when detecting a potential deadlock cycle it will break it by choosing a victim to *abort* and *roll-back*.

Each data versioning, eager and lazy, has its own advantages and disadvantages. Eager versioning systems have a higher overhead on transaction abort because they have to restore the memory changes. In contrast, lazy versioning aborts have smaller overhead since no speculative updates were applied to main

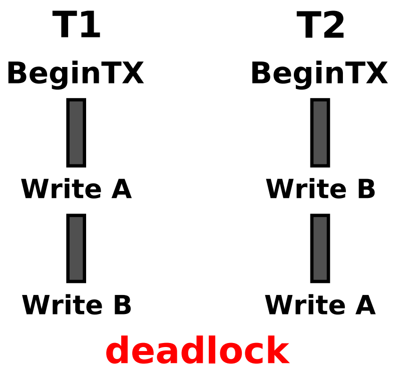


Figure 2.2: Deadlock situation in a system running with an eager data versioning policy. Both transactions are waiting for the other one to free a locked address.

memory. However, lazy policies have performance penalty at commit time, when all transactional updates become visible.

2.2.2 Optimistic versus Pessimistic Conflict Detection

Conflict detection can be performed either taking an optimistic or a pessimistic approach. Systems with *pessimistic conflict detection* notice possible data dependency violations as soon as possible, checking for conflicts on each memory access during transaction execution. On the other hand, *optimistic conflict detection* assumes that a transaction is going to commit successfully and waits until the transaction finishes its execution to detect the conflicts.

Conflict management can substantially affect system performance, we illustrate the problem in Figure 2.3 (this example is inspired from (34)). Pessimistic conflict detection, Figure 2.3-a, attempts to minimize the amount of wasted work in the system. Transaction T_1 conflicts with T_2 and is stalled (waiting for T_2 to finish). Then, T_2 conflicts later on its execution with T_3 and gets stalled too. Note that T_1 now has to wait until T_3 (and then T_2) either aborts or commits, even though it does not conflict with T_3 at all. Most systems that use this pessimistic approach suffer from these so-called *cascading waits* (14), where a transaction is stalled waiting for transactions to finish even if there are no conflicts between them.

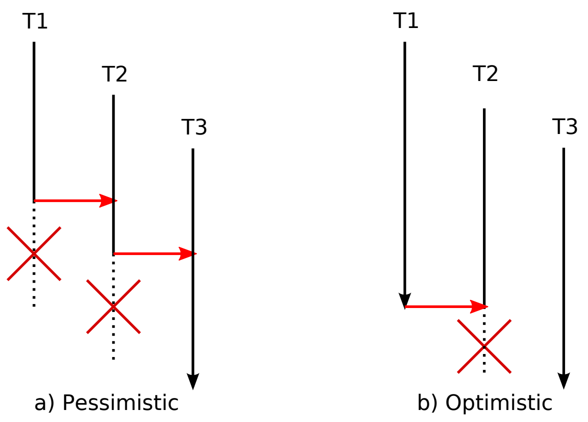


Figure 2.3: Pessimistic and optimistic conflict detection behaviors.

Figure 2.3-b, shows the same execution with optimistic conflict detection. In this case, all the transactions are executing until one reaches its commit point. As we can see, transaction T_1 attempts to commit, and therefore transaction T_2 aborts. Then, T_3 can also commit without conflicts.

As we can see with this example, attempts by pessimistic systems to reduce wasted work are not always successful. In Figure 2.3-a, for instance, the systems stalls T_1, and since it does not eventually abort, the work that was avoided would have been useful. This happens due to a limitation in pessimistic systems; it addresses *potential* conflicts, caused by an offending access to a shared location, at this point they have to speculate which transaction is more likely to commit (and which should be aborted), but the system at this time does not have all the necessary information to make the optimal decision and the prediction is sometimes wrong. On the other hand, optimistic conflict detection deals with conflicts that are *unavoidable* in order to allow a transaction to commit.

Previous research claims that *optimistic conflict detection* allows for more parallelism (14; 24; 31), delaying conflict detection at commit time avoids speculative decisions of which is the best transaction to abort, simplifies the system, and also results in higher performance due to a larger number of transactions committing. Furthermore, optimistic conflict detection guarantees forward progress, processor instructions are guaranteed to complete properly.

2.2.3 Synergistic Combinations

We introduced two ways to deal with data versioning (eagerly or lazily) and two ways to treat conflict detection (pessimistically or optimistically). Intuitively, eager data versioning, where memory updates are done while the transaction is executed, is commonly used with pessimistic conflict detection to ensure that only one transaction has exclusive access to write a new version of a given address. On the other hand, lazy data versioning is usually combined with optimistic conflict detection, doing both tasks (conflict detection and memory updates) at commit time.

However, these are not the only two alternatives. Some of the first TM proposals provide lazy versioning with pessimistic conflict detection. On the other hand, recent research tries to split the monolithic task of conflict detection and adopt an approach that detects conflicts while the transaction is still active (i.e., at every memory access), but resolves them when the transaction is ready to commit (35).

2.3 Software versus Hardware

Software Transactional Memory (STM) systems are not expensive to build, are very flexible about implementing different policies and do not suffer from buffering overflow constrains. However, software is not as fast as dedicated hardware because of the greater overheads it has on every memory access, where data structures must be maintained and eventually queried to perform conflict detection. In contrast, Hardware Transactional Memory (HTM) systems offer higher performance because no software annotations are required on memory accesses. However, they can encounter difficulties when transactionally-accessed lines overflow the capacity of the cache.

Another approach for TM are hybrid systems, like (18). These systems try to either address the challenges of HTM systems switching to an STM system when, for example, hardware resources become exhausted, or even introduce hardware changes to gain performance and address bottlenecks of software transactions.

Chapter 3

Initial Study

This chapter focuses on the explanation of the HTM protocol and the main characteristics and configuration choices of the M5 simulator.

3.1 Scalable-TCC - The Protocol

This section covers the protocol explanation. Firstly, an overview and some protocol details are exposed. Then, we show two detailed examples that clarify the protocol operation process.

3.1.1 Protocol Overview

Scalable-TCC is a non-blocking implementation of TM that is tuned for continuous use of transactions within parallel programs (19). By adopting continuous transactions we can implement a single coherence protocol, we don't need to distinguish transactional accesses from non-transactional since all memory accesses will be considered transactional. This fact makes the consistency model much simpler and easier to understand.

The "illusion" of executing always-in-transaction does not need any code modification at all, it is completely transparent to the programmers point of view and it is handled at runtime by the HTM. When a memory access is executed outside a real or *explicit transaction* (i.e., outside an atomic block, see Figure 2.1), it is detected and the system immediately starts a forced or *implicit transaction* (assuming that no other implicit transaction is running). If the system is executing an implicit transaction and an explicit transaction is going to start, it will automatically commit the first one in order to guarantee the properties of atomicity and isolation of explicit transactions. Note that implicit transactions can be committed at any point in time without altering the correctness of the execution whatsoever.

One of the best properties of this protocol is its non-blocking implementation. This is achieved by detecting conflicts only when a transaction is ready to commit; hence, using *optimistic conflict detection*, running the transaction without acquiring locks, optimistically assuming that no other transaction operates concurrently on the same data. If conflicts between transactions are detected, the non-committing transactions abort, their local updates are rolled-back, and they are re-executed. Scalable-TCC also uses *lazy data versioning* which allows transactional data into the system memory only when a transaction commits. Having lazy data versioning guarantees deadlock and livelock freedom without intervention from user-level contention managers as we mentioned in Section 2.2.1.

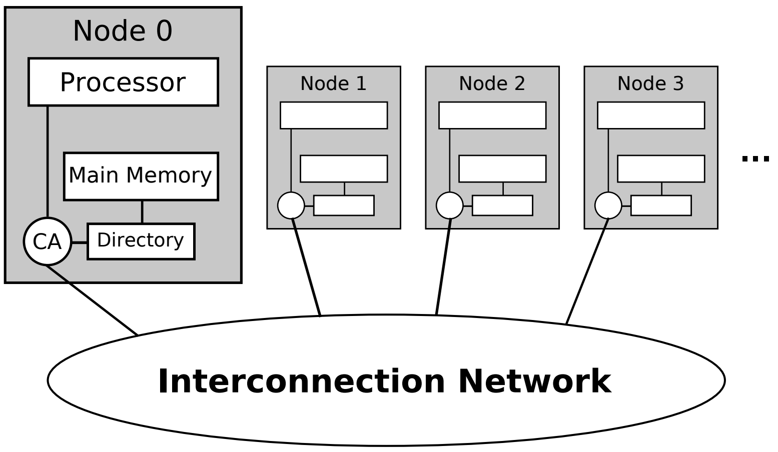


Figure 3.1: System organization schema; CA states for *Communication Assistant*.

Figure 3.1 shows the system organization. We use distributed shared memory (DSM) with directory-based coherence. Using DSM means that the memory address space is split amongst the different nodes of the system, so each memory address belongs to one node. In directory-based coherence, information about the addresses being shared (e.g., a sharers list) is placed in the directories that maintain the coherence between caches. Note that since DSM is used, each directory will hold information of the addresses belonging to its own node. The directory acts like a filter through which the processor must ask permission to load an address from main memory to its caches. We will show details about the directory behavior later in this section; see examples 3.1.3 and 3.1.4.

3.1.2 Protocol Details

The key idea that allows the protocol to achieve a high degree of concurrency is the possibility to commit two or more transactions in parallel if they involve data from separate directories.

First, transactions are executed, *execution phase*, during the execution all memory accesses are not visible to the rest of the system. This means that: a) read and written addresses are added to per-transaction private structures called *read-set* and *write-set* respectively; b) written data is buffered locally in private caches.

Then, the transaction is ready to start the *commit process*, which has two phases:

- **Validation Phase:** The system ensures that the transaction is serially valid. To check this condition the system asks, *only* the directories involved in the *write-set* and the *read-set*, if there are younger transactions waiting to commit on these directories. If there are no younger transactions waiting, this phase completes and the transaction cannot be aborted by other transactions anymore. Note that two transactions with a disjoint set of committing directories can go through the commit process completely in parallel.

- **Commit Phase:** After validation phase, the transaction makes its write-state visible to the rest of the system and the commit process finishes.

Directories are used to track processors that may have speculatively read shared data. When a processor is in its validation phase, it acquires a transactional ID (TID) and does not proceed to its commit phase until it is guaranteed that no other processor can abort it, then sends its commit addresses only to the directories responsible for data written by the transaction. The directories generate invalidation messages to processors that were marked as having read what is now invalid data. Processors receiving invalidation messages then use their own tracking facilities to determine whether to abort or just invalidate the cache-line if it was not read in the current transaction. Note that we use the term *cache-line* here, this is because our HTM implementation works with cache-line granularity, all the addresses present in read or write sets and in the directories are cache-line addresses.

Each directory tracks the TID currently allowed to commit in the *Now Servicing TID (NSTID)* register. When a transaction has nothing to send to a particular directory by the time it is ready to commit, it informs the directory by sending a *Skip* message that includes its TID, so that the directory knows not to wait for that particular transaction. A complete list of the coherence messages used in our Scalable-TCC-like implementation is shown in Table 3.1.

Message	**Description**
Add Sharer	Load a cache-line
TID Request	Request a transactional identifier
Skip	Instructs a directory to skip a given TID
NSTID Probe	Probes for the Now Servicing TID register
Commit	Instructs a directory to commit marked lines
Invalidate	Instructs a processor to treat an invalidation
Abort	Instructs a directory to abort a given TID
Mark	Marks a line intended to be committed
Data Request	Instructs a processor to flush a given cache-line to memory
Write Back	Write back a committed cache-line

Table 3.1: The coherence messages used to implement Scalable-TCC protocol.

3.1.3 Commit Example

The following example, see Figure 3.2, attempts to illustrate all the possible situations that may occur during the execution phase and during a successful commit process between two processors. In the example, inspired from (19), a transaction in P1 successfully commits with one directory while a second transaction in P2 aborts and restarts.

During the explanation of the example we use the coherence messages listed in Table 3.1. Changes in the state are circled and events numbered to show order, meaning all events numbered ❶ can occur at the same time and an event labeled ❷ can only occur after all events labeled ❶ are done.

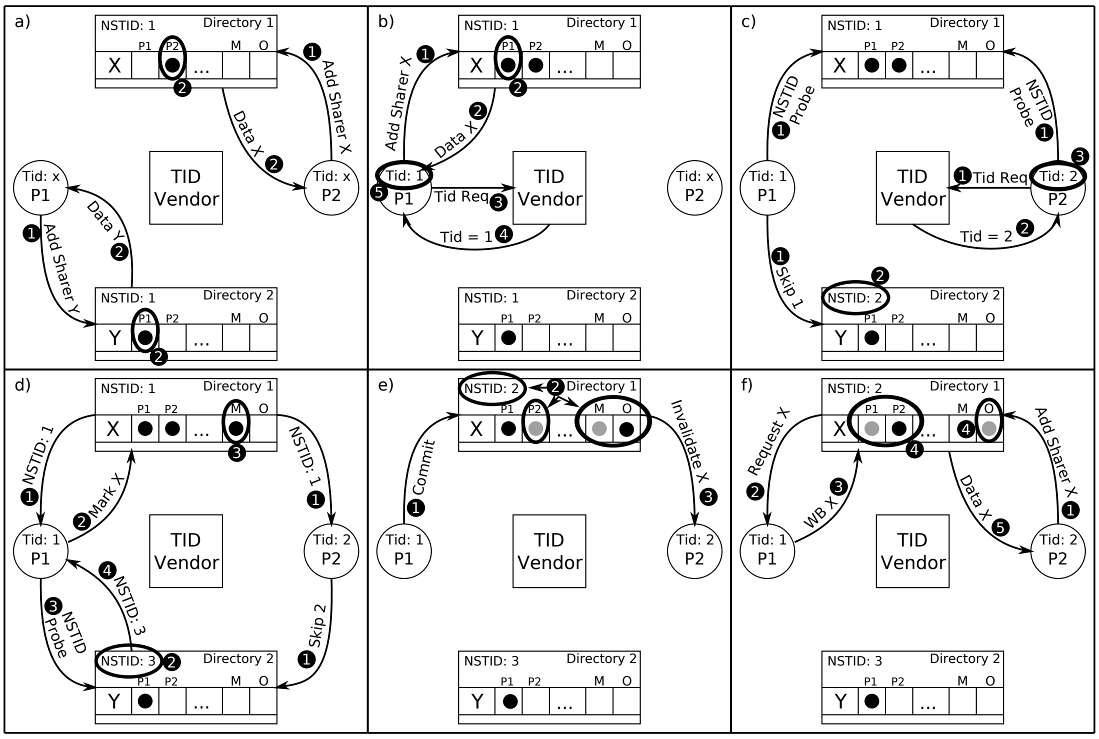


Figure 3.2: Scalable-TCC commit execution example.

In part a, processors P1 and P2 each load a cache-line using the *Add Sharer* message ❶, and are marked as sharers by Directory 2 and Directory 1 respectively. Note that now both processors are in the *execution phase* at the same time.

In this example, both processors write to data tracked by Directory 1, but this information is not communicated to the directory until the *commit phase*. In part b, processor P1 loads another cache-line from Directory 1 and then starts the commit process, thus it starts the *validation phase*. Then, it first sends a *TID Request* message to the `TID Vendor` ❸, which responds with *TID* 1 ❹ and processor `P1` records it ❺.

In part c, `P1`, that is still in its validation phase, communicates with `Directory 1`, the only directory it wants to write to. First, it probes this directory for its *Now Servicing TID (NSTID)* register using an *NSTID Probe* message ❶. In parallel, `P1` sends to `Directory 2` a *Skip* message since `Directory 2` is not in the write-set, causing it to increase its NSTID to 2 ❷. Meanwhile, `P2` has also started the commit process (validation phase). It requests a TID ❶, but can also start probing for `Directory 1` NSTID register ❶— probing does not require the processor to have acquired a TID. `P2` receives *TID* 2 ❷ and records it internally ❸.

In part d, both `P1` and `P2` receive answers to their probing messages, and `P2` also sends a *Skip* message to `Directory 2` ❶ that updates its NSTID register to 3 ❷. `P2` cannot finish its validation phase because the TID answer it received is lower than its own TID. On the other hand, `P1's` TID is equal to the NSTID register from `Directory 1`, thus it can send commit-addresses using *Mark* messages to that directory. `P1` sends a *Mark* message ❷, and line `X` becomes marked (M) as part of the committing transaction's write-set ❸. Mark messages allow transactions to pre-commit addresses to the subset of directories that are ready to service the transaction. In order to finish the validation phase, `P1` has to be sure that no other transactions with a lower TID can violate it. This is done by checking that every directory in its read-set (1 and 2) has finished servicing younger transactions. Since it has already marked lines in `Directory 1`, it can be certain that all transactions with lower TID's have been serviced by this directory. However, `Directory 2` needs to be probed ❸. `P1` receives NSTID 3 as answer ❹, this means that it can be certain that all transactions younger than TID 3 have been already serviced by `Directory 2`. Thus, `P1` cannot be aborted by commits to any directory and finishes the validation phase.

In part e, `P1` sends a *Commit* message ❶, which causes all marked (M) lines to become owned (O) ❷. Each marked line that transitions to owned generates invalidations that are sent to all sharers of that line ❸, except the committing processor which becomes the new owner. `P2` receives the invalidation, discards the line from its private caches and aborts because its current transaction had read it. Note that during the whole commit process no data is communicated between nodes and directories, only addresses, making the process quite cheap in time basis.

In part f, P2, that has to re-execute the transaction, attempts to load an owned line ❶, this causes the directory to send a data request to the owner ❷, the owner then writes back the cache-line and invalidates the line in its private caches ❸. When receiving the data, the directory removes the ownership that P1 had and adds P2 as new sharer for that line ❹, then forwards the data to the requesting processor (P2) ❺.

Each commit requires the transaction to send a single multi-cast skip message to the set of directories not present either in the read or write sets. The transaction also communicates with directories in its write-set, and probes directories in its read-set. Even though this might seem a large number of messages, we will show in Section 9.3 that this communication does not damage performance scalability since the number of directories touch per transaction is small in the common case. Furthermore, we will show also that our implementation scales very well in practice indeed.

3.1.4 Parallel Commit Example

Here we will show two different scenarios. Firstly, a successful parallel commit involving two transactions that have disjoint read and write sets. Secondly, we show the behavior of the system when a transaction that has started the commit process (validation phase) is aborted. Thus, the parallel commit fails and we have to undo the changes done in the directory by the aborted transaction.

Figure 3.3 assumes that both processors have already asked for a TID and were assigned TID 1 and 2 respectively. P1 has written data from Directory 2, while P2 has written data from Directory 1. Note that the only difference between parts a^* and a, is that in part a^* P2 is also marked as sharer of line Y in Directory 2.

To start with, we describe the example exposed in the first row where both processors are able to commit in parallel. In part a, P1 and P2 probe the direc-

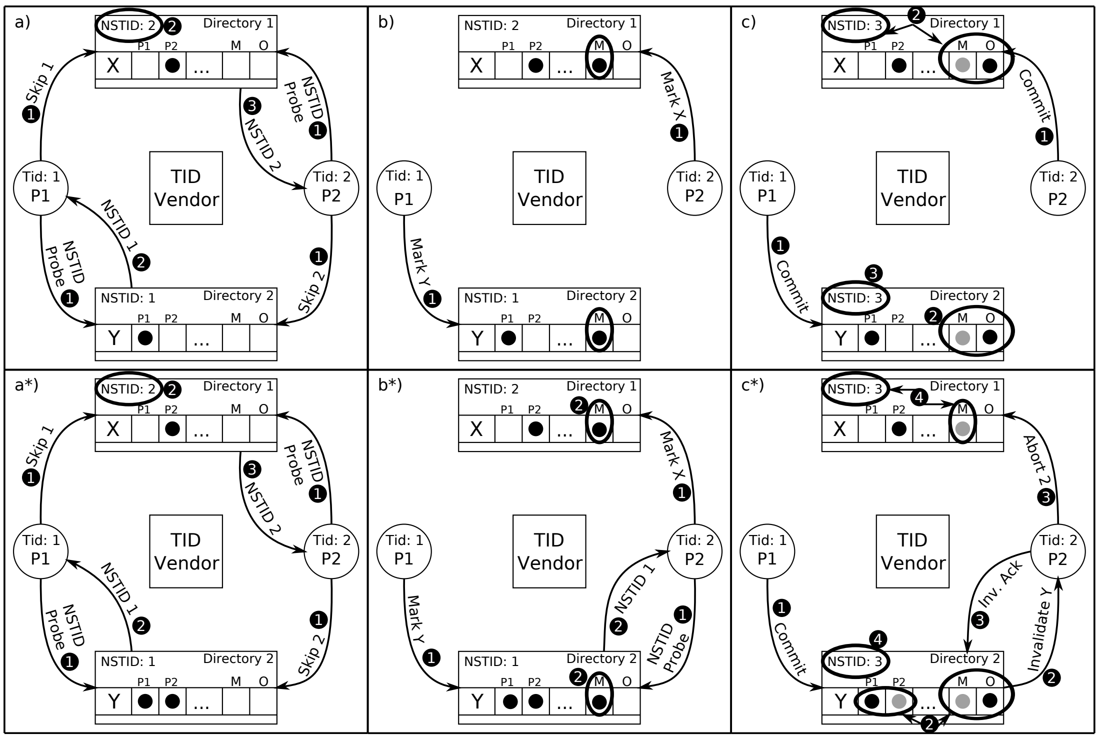


Figure 3.3: Two scenarios attempting to commit a couple of transactions in parallel. Note that in both scenarios the messages generated at the beginning are the same, but in part a) P2 is not marked as sharer of line Y in Directory 2.

tories in their write sets and send the needed *Skip* messages ❶. Both processors receive as answer of the probing a TID that matches their own TID ❷ ❸. Since the *read sets coincide with the write sets* no extra probing is needed and validation phase finishes for both.

In parts b and c, a parallel commit takes place. In part b, both processors send a *Mark* message to the directory where they wrote to; `P1` sends the message to `Directory 2` and `P2` to `Directory 1` ❶. In part c, commit messages are sent ❶ and the directories update concurrently the sharers list, the owner and the NSTID register ❷.

The second row shows an example where parallel commit fails and `P2's` transaction has to abort because it read a line from `Directory 2`, and `P1` will commit there due to its lower TID. In part b*, `P2` has to probe `Directory 2` because is in its *read set* ❶. Meanwhile P2 also sends a *Mark* message to Directory 1 since it is ready to service TID 2. Note that *Mark* messages can be sent before finishing the validation phase. P2 receives an answer of the probing with *NSTID* 1 ❷, which is smaller than its own TID, thus it cannot proceed. It will have to keep probing until the NSTID received is higher or equal than its own. So two commits that involve the same directory, Directory 2 in this case, must be serialized.

In part c*, P1 has finished the validation phase and the marking process, thus it can send the *Commit* messages to the set of directories involved in the write set (i.e., Directory 2) ❶. Once the directory receives the *Commit* message, it generates invalidations to the other sharers of the committed cache-line while updates the ownership ❷. Since line Y was speculatively read by P2, the *Invalidation* message causes it to abort. Since P2 had already sent *Mark* messages, it has to send an *Abort* message to every directory where *Mark* messages where sent, so an *Abort* message is sent to Directory 1 ❸, which causes the directory to clear all the marked bits and to update its NSTID ❹. Note that once the necessary aborts are sent, P2 also needs to send an *Invalidation Ack* message that will cause Directory 2 NSTID register to be updated. This is necessary to avoid certain race conditions; resolves the situation in which a transaction with TID Y is allowed to commit because it received *NSTID Y* as an answer to its probe before receiving an invalidation from transaction X with $X < Y$.

3.2 The M5 Simulator

In order to test our implementation and evaluate performance of the HTM module we will use The M5 Simulator (12), a research tool for computer system architecture widely used by the community. A complete list of publications using this simulator can be found here (4).

3.2.1 Key Features and Configuration Choices

One of the most important features of the simulator and also very important to make it change-prone is its *pervasive object orientation*. All the major simulation structures (CPUs, busses, caches, etc.) are represented as objects, M5's internal object orientation (using C++) provides in addition the usual software engineering advantages. Using a quite simple configuration language that allows flexible composition of this objects we can describe complex simulation targets. This is important for us, because we will have to modify the simulator in order to use our own cache, directory and HTM objects amongst others.

M5 supports multiple interchangeable CPU models, currently there are three different models: a simple in-order CPU; a detailed out-of-order CPU that is superscalar and has simultaneous multi-threading (SMT) capabilities; and a random memory-system tester. The first two models use a common high-level ISA description, we will make some modifications over this ISA to provide new instructions to support transactional executions. We used the *AtomicSimpleCPU*, it is an in-order, one cycle per instruction CPU. This choice is not done to simplify the system, but for consistency with the HTM literature since there are just a few proposals of HTM's using out-of-order CPUs, and as they proved it is quite challenging (36).

M5 features a detailed event-driven memory system, including non-blocking caches over a simple snooping coherence protocol. Since we need a directory-based coherence protocol as shown in the previous section, we have to develop our own implementation for the memory hierarchy (caches, main memory banks and directories). Thanks to M5's object orientation, instantiation of multiple CPU objects within a system is trivial. Combined with our module that will define the memory hierarchy we can easily simulate the desired system.

The simulator supports either full-system and system call emulation execution modes. We are interested in full-system capabilities to be able to have a functional environment able to interact with a disk image for example, since we will store our test binaries there. Full-system mode is only available in Alpha and SPARC architectures. Alpha can boot an unmodified Linux 2.4/2.6 kernel as well as FreeBSD, while SPARC can boot Solaris with some constrains. We chose Alpha architecture to be our testing platform, because using Linux we are sure that all the tests that we will use to evaluate performance and scalability will work properly. Note that no Alpha hardware is needed to make full use of M5 compiled with Alpha architecture, because Alpha binaries to run on M5 can be built on x86 systems using gcc-based cross-compilation tools.

Furthermore, M5 is being released under an open source license. It implies an active community around it with good support from its main developers.

Chapter 4

Related Work

There have been a number of proposals for Transactional Memory (TM) over the last years. In this chapter we will walk through some of them to provide a global view of the research done by the TM community.

TM proposals that use pessimistic conflict detection such as Log-TM (29) and Unbounded TM (UTM) (11), write to memory directly (eager data versioning). This improves the performance of commits, which are more frequent than aborts. However, it may also incur additional violations not present in lazy data versioning. Moreover, UTM tries to address the problem of limited hardware buffering capabilities, by providing mechanisms to support transactions of arbitrary size and duration in a pure hardware approach. However, UTM is not unique in this field, Virtualizing TM (30) provides different mechanisms that shield the programmer from various platform-specific resource limitations.

Scalable-TCC is based on a previous work called TCC (23), it was the first hardware TM system with lazy data versioning and optimistic conflict detection. However, TCC suffers from two major bottlenecks. First, it utilizes an inherently non-scalable communication medium between processors (common bus); and second, all commits are serialized with a commit token which has to be acquired by a transaction at commit time. With Scalable-TCC (19) both problems are addressed.

New proposals are trying to come up with new ideas to take the best of both worlds, lazy-like systems and eager-like systems. Eager-lazy HTM (EazyHTM)

(35) detects conflicts while the transaction is running, but defers conflict resolution to commit time, resulting in a new HTM architecture that performs well, is scalable and easy to implement. Detecting conflicts while the transaction is running makes commit process much faster, and delaying the resolution at commit time does not incur additional violations.

Chapter 5

Architectural Details

In this chapter we discuss the architecture used and the decisions we took about the architectural setup of the whole system. Furthermore, we explain in detail the internals of each main component present in the system.

5.1 Architecture Overview

There are a lot of things to take into account and to decide when setting up such a complex architecture (e.g., interconnection network used, cache hierarchy setup, etc.). Our system is composed of 4 main components: the interconnection network; the directories; the main memory banks; and the processors. As we can see in Figure 3.1 the system is organized in nodes, each one has a directory, a processor and a main memory bank. Nodes communicate with each other through an interconnection network (ICN). In the following sections we explain in detail each component.

5.2 The Processor

Figure 5.1 shows the internals of the processor we are simulating. As we can see in the figure we will use two levels of private data cache (L1 and L2), both tracking the speculatively state of the cache-lines read and/or written by the transactions.

In the CPU side there is a structure called *Register File Checkpoint*. This structure is a replication of the register file present in any CPU. The register file is an array of processor registers. Each architecture has its own set of processor register of different kinds, such as: address registers, which are used by instructions that indirectly access memory; data registers, used to hold numeric values like integers and floating-point values; special purpose registers, including the program counter or the stack pointer; and many others. Thus, the Register File Checkpoint is used to take a snapshot of the current register file state at the beginning of a transaction (i.e., when a begin transaction instruction is executed). In case a transaction aborts, the CPU uses the snapshot to restore its state, this is necessary because we have to restart the execution of the transaction to maintain *atomicity* and we need the CPU to be in the same exact state it was when the transaction started. Note that if we restore the checkpoint, the program counter will point to the first instruction of the aborted transaction (i.e., its begin transaction instruction).

Speculative state is stored in L1 and L2 caches. Note that we do not need non-speculative state tracking since we assume an always-in-transaction scenario, thus all memory accesses are transactional. Figure 5.1 presents data cache organization. Tag bits include *dirty* (D), *valid* (V), *speculatively-read* (SR) and *speculatively-modified* (SM) bits. We have cache-line level speculative state tracking with one bit per field, but word-level tracking is also possible adding more bits to the SR, SM and V fields. In a 32 bytes cache-line configuration, 8 bits per line and per field would be needed.

The SM bit indicates that the corresponding cache-line has been modified during the execution of a transaction. Similarly, the SR bit indicates that its line has been read by a transaction. The valid bit as its name indicates marks invalid data in the cache. Finally, the dirty bit is used to support write-back protocol, we check the dirty bit on the first speculative write in each transaction. If the bit is already set, we first write-back that line to a non-speculative level of the memory hierarchy, in this case, to its associated main memory bank that can be located in any node of the system.

As we said, we will use two levels of private cache. In fact, we need them because we need enough room to store all the speculative memory accesses of a running transaction, and with just one level of cache this would be very difficult even

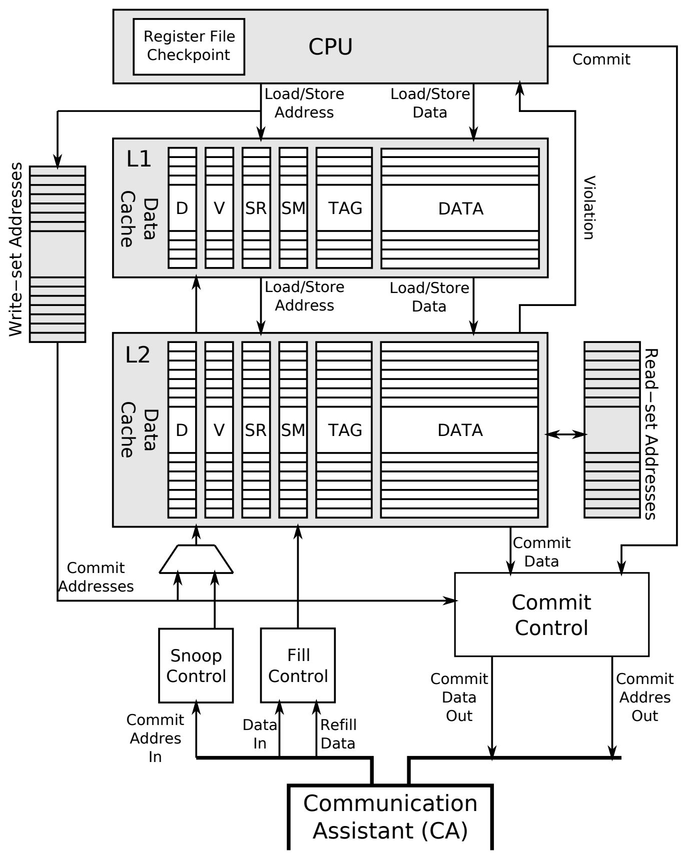


Figure 5.1: Detailed view of the processor internals.

for medium sized transactions since L1 caches tend to be small (∼32KB). However, some researchers have shown that, with relatively large L2 private caches (∼512KB) tracking transactional state, it is unlikely that overflows occur in the common case (21). Moreover, there are mechanisms to deal with the problem with some performance penalties, of course (20; 30).

We will use set associative caches since they provide a good trade-off between hit servicing time and miss rate. Direct mapped caches have the fastest hit times since only one cache position has to be checked to know if a certain line is in the cache, while in a full associative cache all positions have to be checked, but they have the lowest miss rate because any cache-line can be placed to any position (2). As for the replacement policy, when it comes time to load a new line and evict (remove from cache) an old line, we use last recently used (LRU) policy in a per-set basis.

The last two components we want to remark are the *write set* and the *read set*. The first one stores all the written addresses during a transaction of the lines that need to be committed in a FIFO structure. The read set is used to determine weather to violate and abort or just invalidate the line from the caches when an invalidation message is received.

5.3 The Directory

The directory tracks information for each cache-line in the main memory housed in the local node. This information involves a sharers list, a marked bit and an owned bit as shown in Figure 5.2.

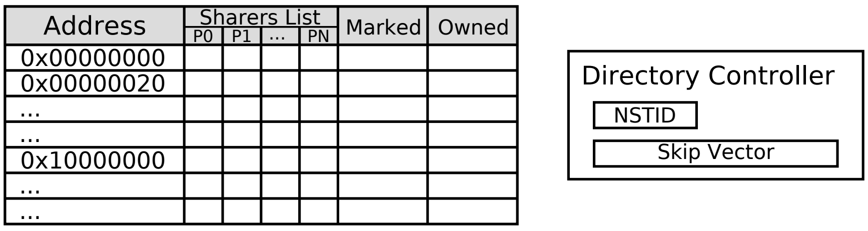


Figure 5.2: Detailed directory structure view.

The sharers list indicates the set of processors that have speculatively accessed the line, so when the line gets committed invalidations will be sent to those nodes. The *Owned bit* tracks the owner for each cache-line, the owner is the last node that committed updates to the line until it writes back to main memory (i.e, an eviction occurs or the line is requested by another node, thus it has to be removed from the private caches). The owner is indicated by setting a single bit in the sharers list and the owned bit. The *Marked bit* is used to indicate lines that are part of an ongoing commit to that directory. Each directory also makes use of a controller that consist in the *NSTID register* and a structure called *skip vector*, that allows the directory to keep track of unordered skip messages received.

Directories control access to a contiguous region of main memory. Only one transaction, at any time, can send state-altering messages to the memory region controlled by that directory, this transaction is the one that has the same TID than the stored in the NSTID register of the directory. If a transaction has nothing to commit to a directory it will send a Skip message with its TID attached. This will make the directory mark the TID as completed. The key point is that each directory will either service or skip every transaction of the system. If two transaction have an overlapping write set, then the affected directory would serialize the commits; in this case, the transaction with the lower TID will always commit first, if the conflicting transaction is not aborted by the first one, then, it can commit later. In other words, a transaction with a higher TID will not be able to write to a directory until all transactions with lower TID have either skipped or committed that directory. Moreover, a transaction cannot commit until all transactions that could abort it have surely finished its execution. This makes the protocol livelock-free and forward process guaranteed.

During the commit process, for each directory that has a satisfactory NSTID, the transaction sends mark messages for the corresponding addresses in its write set. Once marking is complete for all directories involved in the write set and the transaction has received a NSTID higher than its own TID for each directory involved in the read set, the transaction commits by sending a multi-cast *Commit* message to the write set of directories. On receiving this message, each directory will gang-upgrade (all at once) marked lines to owned and generate invalidation messages if there are sharers other than the committing processor. If a transaction after sending mark messages is aborted, it will send abort messages that will make the directories gang-clear mark bits.

5.4 The Interconnection Network

All the messages that have to go from one node to another will use the interconnection network (ICN). We implemented a simple 2-D mesh ICN, as we show in Figure 5.3 with a setup of 16 nodes. We will assume a fixed value of latency per hope and we will always consider the shortest path between two nodes to calculate the cost of sending a message through the network.

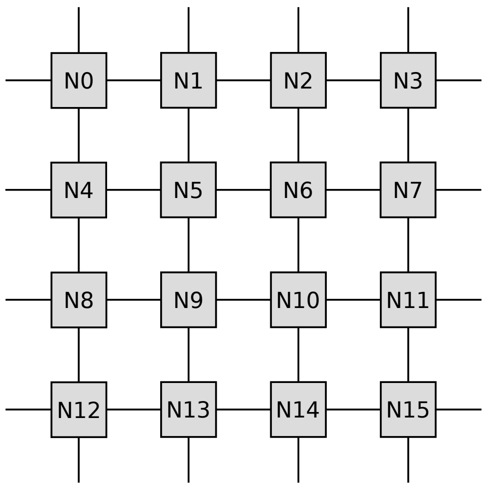


Figure 5.3: Interconnection Network distribution, node organized.

Chapter 6

Cache with Directory Module

In this chapter we introduce the module that allows to handle the definition of the cache hierarchy (L1, L2, etc.), main memory and directory structures using a DSM schema. Statistics are incorporated at cache, main memory and directory level; tracking events such as hits, misses and ticks spent amongst others.

In the following sections, details about the specification and implementation of the cache module are treated. First, we show in detail all the possible transitions that the cache can perform using a diagram, explaining the actions taken on each transition. Followed by the functionalities needed to be implemented for all the structures. Then, we show the possibilities that offer the statistics we implemented. Finally, the chapter concludes with a unit test example and the statistics obtained from its execution.

6.1 Cache State Transitions

In Figure 6.1 we show a diagram of the state transitions that are possible in our cache implementation. It is based on five states that a line in the cache memory can have. These five states are the ones we explained in Section 5.2 plus an extra state, the "dirty and speculatively read" state. Since the hardware allows for combined states set in the caches, we can take advantage of this feature and serve as hits, without writing back, reads over dirty lines.

The diagram assumes only one level of cache, even though it extends to multiple levels, because we always maintain all private cache levels consistent if possible; that is, if a line is in speculatively read state in L1 and L2, and a write takes place, we will access both levels to update data and state to speculatively modified.

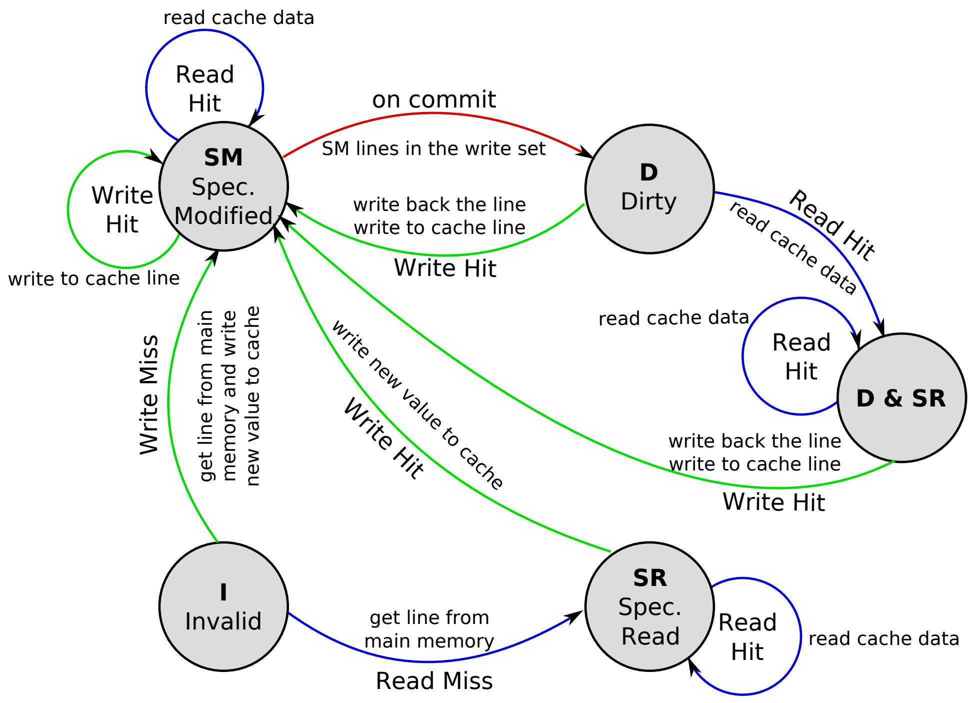


Figure 6.1: Cache transitions state diagram.

Although Figure 6.1 is very clear, here is a brief explanation. We have to remember that any data modification is stored into the cache and must not be visible to the rest of the system. From each state we can either have a read or a write. At the beginning, when the cache is empty and the CPU wants to access a certain line, we will have a cache miss, thus we have to get the line from main memory and the resulting state will be either SM if the access was a write or SR if the access was a read. When in SM state any memory access will be a hit and data read or updated in place. On the other hand, with SR state, if a write takes place we have to change to SM state (updating data in place), but with reads the state remains unchanged. Note that in the graph we show a transition that takes place when committing a transaction, this transition is necessary to show how the dirty state is reached. From dirty state if a write takes place, the line needs to be written back to main memory because it is the only copy of a "visible" portion of data, and then it has to be updated in place with the new value and state changed to SM. Upon a read we change state to dirty and SR which has the same behavior than dirty when a write is executed. We included the dirty and SR state in the diagram for clarity, since when an evict occurs, in both cases, we have to unset the owner in the directory and write back to main memory, but in the second case, a transaction could be aborted because the address might be part of the current transaction; thus, the actions taken are not exactly the same for both states in some cases.

It should be taken into account that the state of a cache line can change because of actions taken also by other CPUs, this is not represented in the graph. Possible scenarios are:

- On receiving an invalidate message for a given address, that line will immediately change its state to invalid. Note that we can never receive an invalidation message for a line in dirty state, because if the line is set as dirty it means that it is owned by the same processor in the corresponding directory.
- On receiving a data request message, in this case the line must be dirty, thus a write back takes place and the line is removed from the cache if it was in dirty state or left as SR if it was in dirty and SR state.

6.2 Configuration and Functionalities

This module implements functionalities of three structures: cache, main memory and directory. Here we will first show the configuration possibilities for these structures and then its main functionalities.

In the cache structure side we can set several configuration parameters, we now enlist the most important ones:

- *Number of cache entries:* A cache entry is a container with an associated direct index. In associative caches an address can only be stored in one of these entries that compose the cache. In Figure 6.2 we can see how an associative cache is filled, this figure shows a cache with two direct entries with index 0 and 1.

- *Associativity:* The associativity of a cache determines how many cache lines can be stored inside a direct entry. In the example of Figure 6.2 the associativity is 2.
- *Cache line size:* As its name indicates this parameter sets the amount of bytes stored per cache line.
- *Cache access cost:* The cost that takes to access the cache and service the request. For example, if we have an L1 cache with 2 cycles access latency and an L2 cache with 10 cycles access cost, it would take 12 cycles to get a cache line stored in L2.

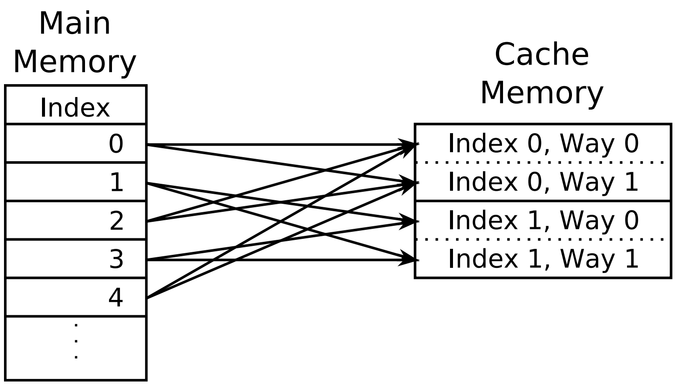


Figure 6.2: 2-Way associative cache fill.

The main memory structure also has some configuration parameters, the following list shows them:

- *Address space size:* Determines the total amount of main memory that the system has distributed amongst all the nodes.
- *Memory access cost:* The cost that takes to access a main memory bank and service the request. Following the previous example, with an access latency of a hundred cycles, getting a cache line from main memory within the same node would cost 112 cycles.

Note that we are not talking about the 2-D Mesh interconnection network. This is because we will not represent it as a structure since we just calculate the minimum number of hops required to access or to reach a certain node from the source, and calculate the latency multiplying the number of hops by a configurable *hop latency.*

Each one of the three mentioned structures (cache, main memory and directory) has plenty of functionalities. Starting with the cache, the following list shows the most important functionalities that have to be implemented.

- *Get Line:* Services a memory access of a given type (read or write), exploring the cache memory hierarchy in ascending order. If a hit occurs, we found the line in the caches; otherwise, we have to forward the request to the main memory level. Considering that accesses at main memory level are always a hit.
- *Service Request:* Services a request started by another CPU over an owned line that is present in *this* cache as dirty. The line is written back and invalidated from the caches if it was just in dirty state, or written back and left as sharer if it was in dirt and SR state.
- *Invalidate Line:* Instructs the cache to set as invalid a given cache-line address.
- *Evict Line:* Evict (remove) a given cache-line address of the cache if present, and also in the lower cache levels if any. When evicting a dirty line we have to write it back. Note that evict is used when there is not enough room to put another line inside the cache, thus evicting a line that is either in the read or the write set of a running transaction would abort the transaction.
- *Line Present:* To know if a line is present in any valid state inside the cache.
- *Get Line Size:* To know the size of the cache-line in *this* cache, in bytes.
- *Line Valid:* Determines if an existing cache-line is in a valid state.
- *Address to Entry:* Function that relates an address with its associated direct entry in *this* cache.

- *Address to Node:* Function that relates an address with its associated node in the system. We take the necessary bits to index the number of nodes in the system starting from the 12th bit. This way we prevent memory pages to be split over different nodes, 2^{12} is equal to 4KB, the usual size of a memory page.

Regarding main memory functionalities, assuming that any access will be considered a hit, we have the following ones:

- *Get Line:* Services a line request that could not be serviced by the caches, we always consider accesses to main memory as being hits.
- *Write Back Line:* Used when a write back to main memory takes place.

Directory functionalities are mainly to service the messages received from any processor through the ICN. The most important are:

- *Add Sharer:* Sets a given processor as being sharer of a given cache-line address in the directory sharers list.
- *Remove Sharer:* Removes the given processor from the sharers list of the given cache-line address.
- *Is Sharer:* Determines if a given processor is sharer of the given line.
- *Is Marked:* Determines if an entry in the directory is set as Marked.
- *Is Owned:* Determines if an entry in the directory is set as Owned.
- *Set Marked:* Sets a given line as Marked.
- *Set Owned:* Sets a given line as Owned. The owner must be one of the sharers of the line.
- *Unset Marked:* Unset a given Marked line.
- *Unset Owned:* Unset a given Owned line.
- *Get Owner:* Used to know the processor that is owning a given owned line.

- *Get Sharers:* Used to know the list of sharers of a given line.
- *NSTIDProbe:* Probes the NSTID register of the directory.
- *Skip:* Updates the skip vector structure and the NSTID register according to the received TID.
- *Commit:* Gang updates (all at once) all the Marked lines to Owned and the directory generates invalidation messages to the sharers of these lines (other than the owner).
- *Abort:* Instructs the directory to unmark Marked lines and update the NSTID register (skip the actual value).
- *Forward request:* Send a cache-line request to the owner processor. Triggered when a processor tries to access a line owned by another processor.

Almost all the functionalities listed also gather and update statistics that will allow us to take conclusions about the different executions that we will show in future sections.

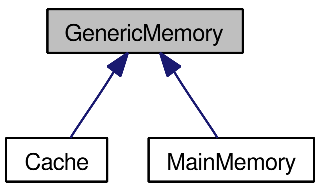
6.3 Diagrams

Herewith we show, to have a general picture, a set of diagrams (Figure 6.3) that illustrate the relations between the components of this module in our implementation. This diagrams have been automatically obtained using *doxygen* (3), which is a very useful documentation system that we used to document the code.

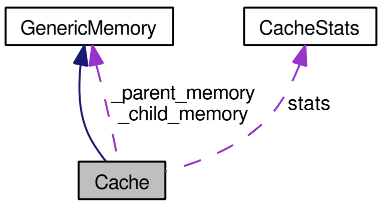
The diagrams show the relations from the point of view of the cache and main memory structures. As shown in Figure 6.3(a) both inherit from a structure called `GenericMemory` that defines common attributes and functionalities.

Figure 6.3(b) shows the relations from the point of view of the cache structure. Each cache has a *parent* and a *child* memory which must be of `GenericMemory` type, that is, either another cache or main memory; obviously for the first level there is no *child* memory. Each cache component has also its own statistics.

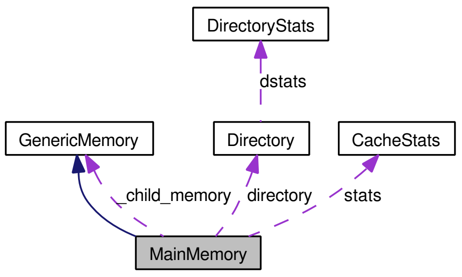
Figure 6.3(c) shows the same schema for the main memory structure, it also has statistics and a *child* generic memory (of cache type in this case), but there is no parent since it is always at the top level. We have defined for convenience the directory inside the main memory structure, which also has its own statistics.



(a) Generic memory inheritance graph.



(b) Cache relations diagram.



(c) Main Memory relations diagram.

Figure 6.3: Inheritance and relation diagrams from cache and main memory point of view.

6.4 Statistics

By having statistics we can use the information gathered to know if the system that we are simulating is behaving as we expected, and we can also be aware of system performance penalties and bottlenecks. Moreover, statistics can be used to fix bugs and other problems related with programming. In this section we show the different statistics that this module collects, we have stats for cache and main memory which use the same interface and statistics for the directory that gather mainly information about the messages received and sent.

6.4.1 Cache and Main Memory Statistics

Figure 6.4 shows the struct that is used to manage the statistics at cache and main memory levels. We use this interface for both. But, for example, when collecting information at main memory level we will never have misses. On the other hand, write backs are tracked only at main memory level.

We can gather information about the number of hits and misses, caused either by a load or a store, at any level of the memory hierarchy; and the number of cycles spent on each level. As we can see in the Figure there is a function to initialize the counters and some functions to count each tracked event as it occurs. The statistics are printed in a file when the execution finishes using the *dump* function.

We show an example of the output file obtained when executing one of the unit tests we used during the development of the module later in this chapter.

```
struct CacheStats {
    size_t ticks;
    size_t hits;
    size_t misses;
    size_t misses_ld;
    size_t misses_st;
    size_t write_backs;

    CacheStats() :
    ticks(0), hits(0), misses(0), misses_ld(0), misses_st(0), write_backs(0)
        {};

    inline void reset() { ticks=0; hits=0; misses=0; misses_ld=0; misses_st=0; write_backs=0; }

    inline void ticks_inc(size_t cnt=1) { ticks+=cnt; }
    inline void hits_inc(size_t cnt=1) { hits+=cnt; }
    inline void misses_inc(size_t cnt=1) { misses+=cnt; }
    inline void misses_ld_inc(size_t cnt=1) { misses_ld+=cnt; }
    inline void misses_st_inc(size_t cnt=1) { misses_st+=cnt; }
    inline void write_backs_inc(size_t cnt=1) { write_backs+=cnt; }

    inline std::ostream & dump(std::ostream &os, const char *prefix, size_t indentation) {
        os << nspaces(indentation).c_str() << prefix << ":\n";
        os << nspaces(indentation+4).c_str() << "Ticks: " << this->ticks << std::endl;
        os << nspaces(indentation+4).c_str() << "Hits: " << this->hits << std::endl;
        os << nspaces(indentation+4).c_str() << "Misses: " << this->misses << std::endl;
        os << nspaces(indentation+4).c_str() << "Misses Load: " << this->misses_ld << std::endl;
        os << nspaces(indentation+4).c_str() << "Misses Store: " << this->misses_st << std::endl;
        os << nspaces(indentation+4).c_str() << "Write Backs: " << this->write_backs << std::endl;
        return os;
    }
};
```

Figure 6.4: Struct that handles the statistics at cache and main memory levels.

6.4.2 Directory Statistics

In Figure 6.5 we show all the information tracked at directory level. It is very important for us to know the directory load since our system is directory based.

The statistics track the number of accesses a certain directory has serviced after an execution, along with the number of messages of each type it received or sent. The resulting stats are dumped in a file using a similar function than the one shown in Figure 6.4. An example of the output can be seen later in this chapter.

```
struct DirectoryStats {
    size_t accesses;
    size_t add_sharer_msg;
    size_t NSTIDProbe_msg;
    size_t mark_msg;
    size_t commit_msg;
    size_t abort_msg;
    size_t skip_msg;
    size_t inval_sent_msg;

    DirectoryStats() :
    accesses(0), add_sharer_msg(0), NSTIDProbe_msg(0), mark_msg(0), commit_msg(0),
    abort_msg(0), skip_msg(0), inval_sent_msg(0)
        {};

    inline void reset() { accesses=0; add_sharer_msg=0; NSTIDProbe_msg=0; mark_msg=0;
        commit_msg=0; abort_msg=0; skip_msg=0; inval_sent_msg=0; }

    inline void accesses_inc(size_t cnt=1) { accesses+=cnt; }
    inline void add_sharer_msg_inc(size_t cnt=1) { add_sharer_msg+=cnt; }
    inline void NSTIDProbe_msg_inc(size_t cnt=1) { NSTIDProbe_msg+=cnt; }
    inline void mark_msg_inc(size_t cnt=1) { mark_msg+=cnt; }
    inline void commit_msg_inc(size_t cnt=1) { commit_msg+=cnt; }
    inline void abort_msg_inc(size_t cnt=1) { abort_msg+=cnt; }
    inline void skip_msg_inc(size_t cnt=1) { skip_msg+=cnt; }
    inline void inval_sent_msg_inc(size_t cnt=1) { inval_sent_msg+=cnt; }

    ...
};
```

Figure 6.5: Struct that handles the statistics for the directories.

6.5 Programming Methodology and Testing

We used a software development technique called *Test-Driven Development* (TDD) (10), that relies on the repetition of a very short development cycle. First, the developer writes a failing automated test case that defines a desired improvement or new function (a unit test), then produces code to pass that test and finally refactors the new code to acceptable standards.

For this purpose we have developed up to 17 unit tests for this module that test many functionalities, such as: cache-line addition and eviction, setting up a memory hierarchy and check that the access times are correct, check that different configuration parameters and cache-line sizes work as expected, check that directory sharers list and state bits are updated properly, and communication between different nodes. Moreover, some stress tests were also used to prove robustness.

```
QT_TEST(testFourSqr)
{
    QT_CHECK_EQUAL(4 * 4, 16);
}
```

Figure 6.6: Unit test example using QuickTest.

To build the unit tests we used a unit testing framework called *QuickTest* (9). It is totally contained within a single header file and its goal is to let you write each test with a minimal amount of code. It is assertion-based, so if a test fails we can know the exact line that makes it fail. An example of usage is shown in Figure 6.6.

6.6 Unit Test and Statistics Output Example

In this section we show an example of one of the unit tests we used and the resulting statistics that we obtained after executing it. Figure 6.7 shows the test example, and Figure 6.8 shows the statistics output for this test.

The test example, named *cache hierarchy simplest*, defines two levels of private cache and the main memory level, thus a single node is defined. As we can see, the first parameter of the constructors (lines 16-18) is an identifier that indicates the node number at which the structures belong; followed, in the caches definitions, by a string that names them, and a pointer to the parent memory in the hierarchy. The rest of the parameters allow for configuration and the chosen values can be seen in the figure, lines 2-14.

At line 23 we are executing the first memory access at address `A_addr`, asking for SR state, the number of cycles that will be needed to service the request is put in the `num_ticks` variable. Thus, after the execution of the access we can check with one of the macros provided by *QuickTest* if the servicing time is the expected one. Since the caches are empty at this point, the access is a miss in both `L1` and `L2` so accessing to main memory is needed. The time of accessing the directory is included in the time of accessing main memory, so the access time should be, as we can see in line 24, the sum of accessing all the memory levels, that is, 112 cycles. If any of the *QuickTest* checks fails, an error message is printed indicating the line where the error takes place.

At line 27 we access again the same location, but now the address is in both `L1` and `L2` caches, meaning that a hit at `L1` level occurs, and thus the cost of the access is 2 cycles. Next access is done at line 31, where we are accessing a new location that is going to be stored in the same direct entry than address `A_addr`. We can ensure this because we are accessing a location that is the result of adding to `A_addr` the entry cycle, which is the size of the line multiplied by the number of entries. This means that now this direct entry is full in `L1` since its associativity is 2. Similarly than in the first access, this one takes 112 cycles to complete.

At line 35 we are accessing again a new location that has to be stored in the same direct entry. Since it is full, this will effectively evict the last recently used cache-line (`A_addr`). Note, however, that the evicted cache-line is still in `L2` due to its higher associativity. To test that `A_addr` is still in `L2` we access this location again at line 39, checking that its access cost is the sum of `L1` and `L2` costs, 12 cycles.

To finish with the test example, at line 42 we print the statistics to a file using the *dump_stats* function. Note that it is called just at main memory level, but it prints the stats of all the levels recursively. The output of the statistics is shown in Figure 6.8.

```
QT_TEST(cache_hierarchy_simplest) {
    Addr addr_space = 8*GB;
    size_t mem_access_cost = 100;

    size_t L1_direct_entries = 4;
    size_t L1_cl_size_bytes = 64;
    size_t L1_associativity = 2;
    size_t L1_hit_cost_ticks = 2;
    size_t L1_entry_cycle = L1_cl_size_bytes * L1_direct_entries;

    size_t L2_direct_entries = 128;
    size_t L2_cl_size_bytes = 128;
    size_t L2_associativity = 8;
    size_t L2_hit_cost_ticks = 10;

    MainMemory main_mem(0, addr_space, mem_access_cost);
    Cache L2(0, "L2", &main_mem, L2_direct_entries, L2_associativity,
        L2_cl_size_bytes, L2_hit_cost_ticks, IS_TOPLEVEL_CACHE);
    Cache L1(0, "L1", &L2, L1_direct_entries, L1_associativity, L1_cl_size_bytes
        , L1_hit_cost_ticks, !IS_TOPLEVEL_CACHE);

    const Addr A_addr = 100000000;

    size_t num_ticks = 0;
    L1.cl_get(A_addr, STATE_S, num_ticks);
    QT_CHECK_EQUAL(num_ticks, mem_access_cost+L2_hit_cost_ticks+
        L1_hit_cost_ticks);

    num_ticks = 0;
    L1.cl_get(A_addr, STATE_S, num_ticks);
    QT_CHECK_EQUAL(num_ticks, L1_hit_cost_ticks);

    num_ticks = 0;
    L1.cl_get(A_addr+L1_entry_cycle, STATE_S, num_ticks);
    QT_CHECK_EQUAL(num_ticks, mem_access_cost+L2_hit_cost_ticks+
        L1_hit_cost_ticks);

    num_ticks = 0;
    L1.cl_get(A_addr+2*L1_entry_cycle, STATE_S, num_ticks);
    QT_CHECK_EQUAL(num_ticks, mem_access_cost+L2_hit_cost_ticks+
        L1_hit_cost_ticks);

    num_ticks = 0;
    L1.cl_get(A_addr, STATE_S, num_ticks);
    QT_CHECK_EQUAL(num_ticks, L2_hit_cost_ticks+L1_hit_cost_ticks);

    main_mem.dump_stats("Test 8 stats");
}
```

Figure 6.7: Unit test example, cache hierarchy simplest.

The statistics gathered should match with our explanation above. If we sum the number of hits and misses at L1 level we obtain the total amount of accesses done, 5 in this case. There was only one hit at this level, the one done by the access at line 27; therefore, the rest of the accesses were misses. Since one access was serviced by L1 there were 4 accesses that reached the L2 level, again one of them was a hit (line 39), and the rest were misses that had to query the main memory level. Since we access to 3 different locations there were 3 messages to the directory of type *Add Sharer*. Note that we count timings in a cumulative way on each level, so in L1 we count also the time it takes to the other levels in the hierarchy to service the requests, thus we know the total amount of cycles it took to service all the accesses, 350 cycles, of which 300 were spent just on accesses to the main memory level.

```
Statistics:

    # Test 8 stats
    - main_mem:
        Ticks: 300
        Hits: 3
        Misses: 0
        Misses Load: 0
        Misses Store: 0
        Write Backs: 0
    - directory:
        Accesses: 3
        Add Sharer messages: 3
        NSTIDProbe messages: 0
        Mark messages: 0
        Commit messages: 0
        Abort messages: 0
        Skip messages: 0
        Invalalidations sent messages: 0
        L2:
            Ticks: 340
            Hits: 1
            Misses: 3
            Misses Load: 3
            Misses Store: 0
            Write Backs: 0
            L1:
                Ticks: 350
                Hits: 1
                Misses: 4
                Misses Load: 4
                Misses Store: 0
                Write Backs: 0
```

Figure 6.8: Cache statistics results of executing the cache_hierarchy_simplest test.

Chapter 7

The HTM Module

In this chapter we introduce the HTM module, this module uses the cache with directory module to define the memory hierarchy. This module mainly defines the concept of transaction and the HTM interface.

In the following sections some details about the specification and implementation of the HTM module are discussed. First, we show a diagram of the possible state transitions a transaction can do, explaining the insights of each case. Followed by a list of the functionalities needed to be implemented in all the structures. Then, we show the relation diagrams and the possibilities that offer the statistics we implemented. Finally, the chapter concludes with a unit test example and the statistics obtained from its execution.

7.1 HTM State Transitions

In Figure 7.1 we show the state transition diagram for our HTM implementation. Each transaction that is running in a processor has its own associated state (along with other information). Thus the processor knows, depending on the current transaction state, what to do. We will now explain in detail the actions that are taken depending on the state of a transaction.

The initial state is the *free* state, even though, technically, code is never executed while in free state. This is because we use the already mentioned always-in-transaction approach. Meaning that every instruction has to be part of a transaction, except for begin transaction instructions that will change the state to *active*. If any other kind of instruction is about to be executed while in free state, a begin instruction is immediately executed before, thus an *implicit transaction* starts.

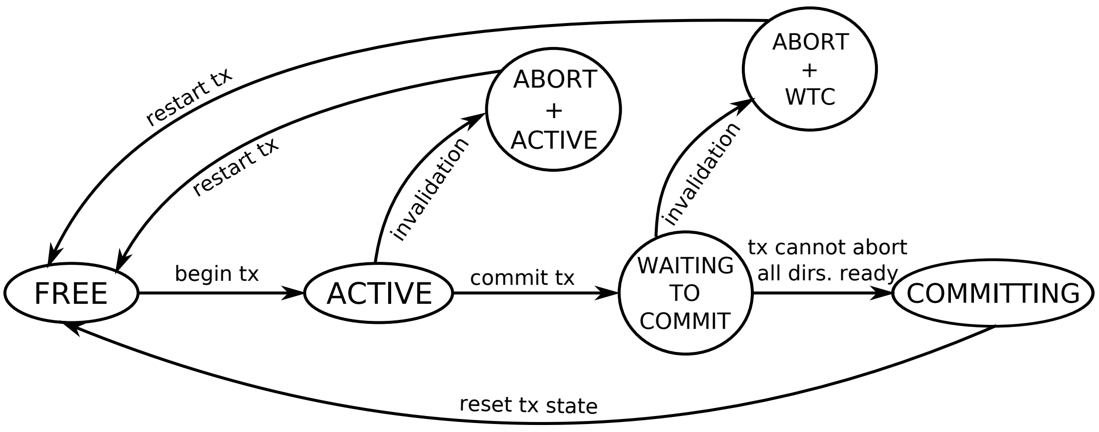


Figure 7.1: HTM state transition diagram. *WTC* stands for waiting to commit

While in *active* state, both *implicit* and *explicit* transactions execute their code normally. Two state-altering situations can occur when in active state: 1) **receive an invalidation message** over a line that is part of the transactions read set, then a conflict is detected and the transaction has to be aborted. Thus, the transaction moves to *abort+active* state. Since the transaction had no associated TID, will ask the vendor to acquire one. Ones the TID is received, a broadcast *Skip* message is sent to instruct the directories to update its skip vector and NSTID register, and the lines involved in the transaction are discarded from private caches. Finally, the register file is restored and the transactional information, including the state, is cleared. 2) **execute a commit transaction instruction**, which will make the transaction ask for a TID and the state will change to *waiting to commit.*

Ones in *waiting to commit* state, the processor starts the validation phase to ensure that the transaction is serially valid by probing the directories like we explained in Section 3.1. When all the directories are ready to service the transaction, it reaches the *committing* state. As happened in active state, the transaction can receive invalidations from other processors. If the invalidation makes the transaction abort it moves to *abort+wtc* state. The aborting process is similar, but in this case, acquiring a TID is not needed since the transaction already has one, and *Abort* messages are sent to the directories where *Mark* messages were sent in the previous state.

After reaching the *committing* state the transaction can no longer be aborted, and we are sure it will commit successfully. In this state the commit phase takes place, *Commit* messages are sent to the writing set of directories to change marked lines to owned, and these lines are also set as dirty in private caches. Now, the commit process has finished, transactional information is cleared, and the next transaction can start its execution following the same transitions.

7.2 Functionalities

This module introduces the concept of transaction, the definition of the HTM interface and the TID Vendor, along with a “fake” processor implementation to be able to test the module.

The processor will use the cache with directory module to define the memory hierarchy, and has some wrappers to functions that interact mainly with the caches. The TID Vendor objective is to service TID requests giving the next ID available.

A transaction has, as we already know, an ID that is unique and a state associated. Furthermore, we need to have for each transaction its read and write sets of addresses. So, basically the related functionalities are focused on interacting with these parameters, such as: get and set a TID of a transaction; state-consulting functions, for example, to know if the transaction is in free state; state-altering functions, to change the transaction state; or to know if a certain line is present in the read or the write set.

The HTM interface defines the new additional behaviors that a processor needs to use in order to make transactional executions possible. Through the interface the processor has access to the transaction information, so it can make use of the transaction functionalities mentioned above. A list of these additional behaviors and its description follows:

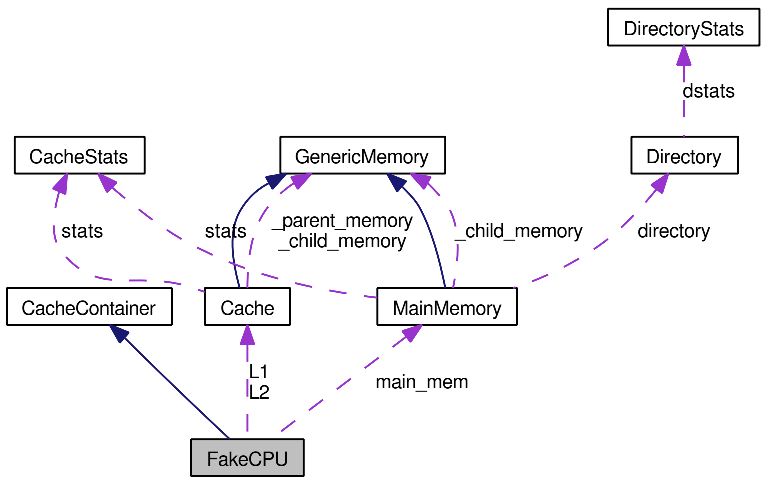
- *Is HTM ready:* Checks if a processor can proceed to the next instruction, this means that this will be done in a per-cycle basis, considering that the number of instructions executed per cycle (IPC) equals one. First, we check if there are pending invalidations to be treated, this operation may abort the transaction by changing the state of the transaction to any of the possible aborting states, depending on the actual one. If there were invalidations we have spent some time treating them, so we have to count it by adding this delay to the processor cycles counter.
  Then, after invalidations are treated, we check the state of the transaction to take the appropriate action. If the state is either free or active, there is nothing else to be done, we let the next instruction start its execution. If the state is ready to commit, we will start probing directories (validation phase), when all are ready we will change the state to committing to finish the commit processes with the commit phase. Similarly, the appropriate actions are taken for the aborting states, which were explained in the previous section. The processor is delayed as many cycles as needed for the system to finish the actions required for each state.
  We must not execute all the actions related to one state as if they were an atomic task, since it would be unrealistic. For example, a processor should be able to receive and process invalidations while executing the validation phase. Thus, we need a way to allow a good degree of concurrency. Lets say we are starting the validation phase, one round of Probing and Skip messages is sent (considering that this is an atomic task) and the appropriate delay is added. At this point, since there is more work to be done in the validation phase, we prevent the processor to execute the next instruction of the running program and we do not allow it to update the program counter either. On the next cycle, the processor will try to execute the instruction again, but it first checks if the HTM is ready, and since it is not, another round of messages is sent, the instruction is not executed and the program counter remains the same. This is done until the commit process finishes. By fractioning the commit process we allow invalidations, or other events, to be treated every time we check if the HTM is ready.

- *Begin transaction:* Starts a new transaction, the transaction must be in free state and changes to active state. A register file checkpoint is done as well.
- *Commit transaction:* The transaction must be in active state, it acquires a TID from the Vendor and sets its state to ready to commit. The rest of the work is delegated to the *is HTM ready* functionality to allow for concurrency.
- *Read transactional:* Is used whenever an instruction needs to execute a read in the memory hierarchy. Essentially, we use the already explained functionality *get a cache-line*, that we have in our cache structure. We add the line address to the read set and then select the bytes from the line that the instruction needs. The processor is delayed as many cycles as it took to get the line from the hierarchy.
- *Write transactional:* Similarly than in the read, we get the line and then we write the new value leaving the modified line in private caches.
- *Terminate transaction:* Either if the transaction committed successfully or aborted we use this functionality to prepare the needed structures and to update the statistics.
- *Is part of transaction:* To check if a given address is either part of the read or the write set.
- *Broadcast Skip:* Sends a Skip message to all the directories of the system for a given TID.
- *Abort all:* Sends an Abort message to the directories that have marked lines of an aborted transaction.
- *Commit all:* Sends a commit message to the writing set of directories.

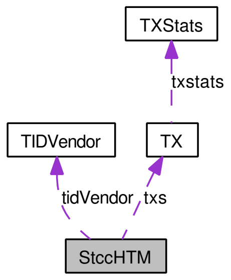
As happened in the previous module, the majority of these functionalities gather and update statistics. It are shown in Section 7.4.

7.3 Diagrams

In Figure 7.2 we show a couple of diagrams that allow us to see all the relations between the different components of the module, including the components of the cache with directory module, to see how they are used.



(a) FakeCPU relations diagram.



(b) Scalable TCC interface relations diagram.

Figure 7.2: Relation diagrams from FakeCPU and HTM interface point of view.

In the first diagram, Figure 7.2(a), we can see how the cache with directory module is used by the FakeCPU structure. The diagram shows that it defines two levels of private cache (L1 and L2) and main memory. We will consider each FakeCPU as a node of the system, thus, for each FakeCPU defined, the system has an extra node that affects the ICN topology and the distribution of the memory address space amongst the nodes.

The second diagram, Figure 7.2(b), shows the relations from the point of view of the HTM interface. We should consider the interface as an extension of functionality, available for all the processors. The interface can communicate with the TID Vendor to obtain a TID when a transaction needs it. It can also access and modify the transactions information, like its state, read set, write set, etc. Statistics per-transaction basis are also collected.

7.4 Statistics

As we explained in the Section 6.4, statistics are very helpful to evaluate the system and to make it error-free. In this section we show the statistics collected for each transaction executed on the system and the resulting global HTM statistics.

7.4.1 Transaction Statistics

In Figure 7.3 we show the statistics that are gathered at transaction level, lines 2-17. So, for each TID we know the processor that executed the transaction, if it committed successfully or aborted, and if it was an implicit or an explicit transaction. Moreover, we also know the read and write set sizes, the number of invalidations received from commits done by other transactions, the number of cycles spent in the ICN by the messages sent to any directory, and the amount of directory messages of each type sent. Finally, we also know the time when the transaction started and its execution time in cycles.

We also show, as example, one of the functions that update the statistics of the Add Sharer directory message, lines 21-25. Note that we are updating here at the same time the statistics at global HTM level and we make distinction between implicit and explicit transactions as we explain in the following section.

This stats are printed in a dedicated file formated with *tab separated values* (TSV), easily exportable to a spreadsheet-like environment.

```
struct TXStats {
  int txid;
  int cpuid;
  bool committed;
  bool real;
  size_t rs_size;
  size_t ws_size;
  size_t invalidation_msgs_recived;
  size_t dir_msgs_cycles;
  size_t dir_msgs_addsharer;
  size_t dir_msgs_nstidprobe;
  size_t dir_msgs_skip;
  size_t dir_msgs_mark;
  size_t dir_msgs_commit;
  size_t dir_msgs_abort;
  size_t btx_time;
  size_t exec_time;

  ...

  inline void dir_msgs_addsharer_inc(size_t cnt=1) {
      dir_msgs_addsharer += cnt;
      if (real) htm_stats_global.real_dir_msgs_addsharer_inc(cnt);
      else htm_stats_global.forced_dir_msgs_addsharer_inc(cnt);
  }

  ...

};
```

Figure 7.3: Statistics gathered at transaction level.

7.4.2 Global HTM Statistics

In Figure 7.4 we show all the information we track at global HTM level. We consider that all the necessary information we would need to extract conclusions is included. In order to know the impact of having an always-in-transaction execution we have separated stats for implicit and explicit transactions. Referred in the figure as forced for implicit and real for explicit.

We track the number of begin and commit transaction instructions, the number of commits that finished successfully, the time spent during commit processes, and the aborts due to conflicts; for both, implicit and explicit transactions. Following the layout of the transaction statistics we also have global stats for the

```
size_t begin_instr;
size_t commit_instr;

size_t real_begin_instr;
size_t real_commit_instr;
size_t real_commit_finished;
size_t real_commit_cycles;
size_t real_aborts_conflicts;
size_t forced_begin_instr;
size_t forced_commit_instr;
size_t forced_commit_finished;
size_t forced_commit_cycles;
size_t forced_aborts_conflicts;

size_t real_dir_msgs_cycles;
size_t real_dir_msgs_addsharer;
size_t real_dir_msgs_nstidprobe;
size_t real_dir_msgs_skip;
size_t real_dir_msgs_mark;
size_t real_dir_msgs_commit;
size_t real_dir_msgs_abort;
size_t forced_dir_msgs_cycles;
size_t forced_dir_msgs_addsharer;
size_t forced_dir_msgs_nstidprobe;
size_t forced_dir_msgs_skip;
size_t forced_dir_msgs_mark;
size_t forced_dir_msgs_commit;
size_t forced_dir_msgs_abort;

size_t real_total_rd;
size_t real_total_wr;
size_t real_total_rd_wasted;
size_t real_total_wr_wasted;
size_t forced_total_rd;
size_t forced_total_wr;
size_t forced_total_rd_wasted;
size_t forced_total_wr_wasted;

size_t invalidation_msgs_recived;
size_t aborts_evictions;

size_t real_cycles_in_txs_total;
size_t forced_cycles_in_txs_total;
size_t diffcurTick;
size_t diffcurTick_parallel;
size_t diffcurTick_barrier;
```

Figure 7.4: Statistics gathered at HTM level.

number of cycles spent in the ICN by the messages sent to any directory, and the amount of directory messages of each type sent during the entire execution. We also count the number of reads and writes done in both implicit and explicit transactions, and the wasted amount of work due to aborts in this sense.

As in transaction level, we can know the number of invalidation messages received, this is an indicator of the amount of shared data used during the execution; the higher this value is, the more accesses to shared data are done during the execution. Two transactions accessing the same line always generate an invalidation message when one of the transactions commits. Aborts caused by evictions should not occur in any case, it would mean that we overflown the private cache capacities and it could lead to an infinite abort cycle over the same transaction, thus by looking at the statistics at runtime we have a mechanism to detect this behavior, that would abort the execution if needed.

Finally, regarding timings, we count the time that the execution spends in implicit and explicit mode in all threads; and the times that will allow us to know if the system scales are: *diffcurTick*, that tells us the total amount of time required to execute the test; *diffcurTick_parallel*, is the time spent in the parallel section, we will use this one to make the scalability study in Section 9.3; and *diffcurTick_barrier*, which is the time spent in barriers if any, useful to justify the behavior of some executions that contain barriers.

As happened with transaction statistics this are also printed in a separated file.

7.5 Unit Test and Statistics Output Example

We followed the same methodology than in the cache with directory module. So, we have implemented a set of unit test using *QuickTest* directives and the test driven development approach to develop this module. The unit tests check from correct state transitions according to specific events to potential circular lock cases, tri-conflict situations, or the so-called always-in-transaction approach.

The FakeCPU processor does not support any kind of timing measurements, thus we will show a simplified version of the statistics output as example, that will show the relevant information for this test execution. Full use of the statistics system is done in the final evaluation.

Figure 7.5 shows one of the unit tests we used to verify our module correctness. It assumes a system with 32 processor (or nodes) defined, but we are using just one of the processors at random. First, at line 8 we start a transaction with the begin transaction instruction on the randomly picked processor. Then we read the variable `A` from memory which is equal `100` and we store the read value in the auxiliary `Ard` variable. Note that we are using the programs memory space as main memory for this tests, and that this read operation loaded into our cache structures the appropriated line. Now, both `A` and `Ard` are equal `100`. Before proceeding with the next instruction we check if the HTM is ready, since it is in active state we can move on. At line 13, we perform a write over the variable `A` with value `101`, but this change is done only in private caches; thus, it is not visible at program's memory space. To check it, at line 14, we make a comparison that verifies that variable `A` is still equal `100`. After checking that we can continue with the next instruction (line 15), we commit the transaction. This action will again not make the line visible, it will be set as dirty in private caches and as owned in the correspondent directory, verified by the check in line 20. The commit instruction can take some time to finish the commit process, so we wait until the HTM is ready again (line 18).

At line 23 a new transaction is started on the same processor. Next, we proceed to read again variable `A` leaving the read value in `Ard`, this read changes the line state from dirty to dirty and SR state in both of the private caches. Note that now we read `101` (line 27), since we got the value directly serviced as a hit from the private caches, where the updated copy of the data resides. Now, we execute a write over variable `A` (line 30), which will effectively cause a write back of the actual data residing in the private caches, making variable `A` change its value to `101` in the program address space (checked at line 31). As final observation, the value residing in the private caches at the end of the test execution is `110`, as a result of the last write executed that changed the value right after the write back.

The statistic outputs are shown in Table 7.1 and Figure 7.6, the ones in the table are at transaction level and the ones in the figure are at global HTM level. In

```
QT_TEST(simple_write_back_tx) {
    int cpuid = (int)(rand() % MAX_CPUIDS);
    unsigned int A=100;
    uint64_t Ard=0;
    Addr refA = (Addr) &A;
    const unsigned int origA = A;

    QT_CHECK_EQUAL(HTM.begin_tx(cpuid), true);
    QT_CHECK_EQUAL(HTM.read_tx(cpuid, refA, sizeof(A), &Ard), NoFault);
    QT_CHECK_EQUAL(A, Ard);
    QT_CHECK_EQUAL(HTM.is_htm_ready(cpuid), NoFault);

    QT_CHECK_EQUAL(HTM.write_tx(cpuid, refA, sizeof(A), Ard+1), NoFault);
    QT_CHECK_EQUAL(A, origA);   // still not visible
    QT_CHECK_EQUAL(HTM.is_htm_ready(cpuid), NoFault);

    QT_CHECK_EQUAL(HTM.commit_tx(cpuid), true); //This does not write back!
    while (HTM.is_htm_ready(cpuid) != NoFault) {};

    QT_CHECK_EQUAL(A, origA);   // still not visible at main memory level!
    QT_CHECK_EQUAL(HTM.is_free(cpuid), true);

    QT_CHECK_EQUAL(HTM.begin_tx(cpuid), true);
    QT_CHECK_EQUAL(HTM.is_htm_ready(cpuid), NoFault);

    QT_CHECK_EQUAL(HTM.read_tx(cpuid, refA, sizeof(A), &Ard), NoFault);
    QT_CHECK_EQUAL(Ard, origA+1);
    QT_CHECK_EQUAL(HTM.is_htm_ready(cpuid), NoFault);

    QT_CHECK_EQUAL(HTM.write_tx(cpuid, refA, sizeof(A), Ard+9), NoFault);
    QT_CHECK_EQUAL(A, origA+1);
    QT_CHECK_EQUAL(HTM.is_htm_ready(cpuid), NoFault);

    QT_CHECK_EQUAL(HTM.commit_tx(cpuid), true);
    while (HTM.is_htm_ready(cpuid) != NoFault) {};
}
```

Figure 7.5: HTM module unit test example, called: simple write back transactional.

the transaction statistics each row means an executed transaction. Since the unit test executes two transactions we have two rows, each one showing a simplified version with the relevant information. We can see that processor 26 executed the transactions with TID 1 and 2 respectively. Both transactions committed successfully and were explicit. Regarding the messages sent to the directory, we can see that 31 Skip messages were sent per transaction, they were sent at commit time, one for every other node on the system, as we stated before we are considering a system with 32 nodes. On the other hand, just one Probe message is sent since we are committing lines from a single directory. Note that the directory message cycles is not time spent by the processor, it is the time that the ICN is working to deliver the messages, and some of them do not make the processor wait for an answer (e.g., Skip messages).

At HTM level (Figure 7.6), the statistics are usually divided in three sections: the first one has general information and combines the information from both explicit and implicit transactions; the second one has all the information related with explicit transactions; and the third one has information regarding implicit transactions, not shown in the figure because no implicit transactions were executed. As we already pointed out, processor clock dependent values are not available at this stage.

Proc	TID	Commited	Real	RS	WS	Dir. MSGs	Dir. MSGs cycles	Add Sharer	Probe	Skip	Mark	Commit
26	1	1	1	1	1	35	1250	1	1	31	1	1
26	2	1	1	1	1	35	1250	1	1	31	1	1

Table 7.1: Transaction stats information, some irrelevant fields were dropped.

```
Statistics:
    HTM statistics TOTAL:
      Begin instr: 2
      Commits started: 2
      Commits finished: 2
      Commit cycles (total): 0
      Aborts: 0
      Dir. messages: 70
      Inval. messages reciv: 0
      Total TXnal reads: 2
      Total TXnal writes: 2
      Tot. TXnal reads wasted: 0
      Tot. TXnal writes wasted: 0
      curTick diff: 0
      curTick parallel diff: 0
      curTick barrier diff: 0
      Cycles in TXs (all thr): 0
    HTM statistics EXPLICIT TX:
      Begin instr: 2
      Commits started: 2
      Commits finished: 2
      Commit cycles: 0
      Aborts evictions: 0
      Aborts conflicts: 0
      Dir. messages: 70
      Dir. messages (cycles): 2500
      Dir. messages addsharer: 2
      Dir. messages nstidprobe: 2
      Dir. messages skip: 62
      Dir. messages mark: 2
      Dir. messages commit: 2
      Dir. messages abort: 0
      TXnal reads: 2
      TXnal writes: 2
      TXnal reads wasted: 0
      TXnal writes wasted: 0
      Cycles in TXs (all thr): 0
```

Figure 7.6: Global HTM statistics, implicit transactions section not shown. Timings are not available.

Chapter 8

Merging within M5 - The Process

This chapter explains the necessary steps to run M5 in Full-System mode, the merging process and the problems we encountered.

8.1 Running M5 in Full-System Mode

First of all, we needed to build the M5 binary for Alpha architecture and Full-System (FS) mode. To be able to run the simulator in FS mode we need a set of files such as a disk image (filesystem), and pre-compiled binaries that include, for example, the Linux kernel we will boot. All these files are available from the M5 site (5). Once everything is set we can launch the simulator using the provided FS configuration and the Linux kernel should boot successfully.

At launch time we can specify a script to be executed just after the kernel boots, if no script is specified the simulator waits for commands to be entered in the terminal. To automate the process of executing the tests we will use bootscripts to run the tests, as the one shown in Figure 8.1, that will be called from another script that will launch all the executions while saving the statistic files.

Note that in Figure 8.1 we are using a path (`htm/st/genome`) that is inside the filesystem of the simulator. We modified the filesystem image to include our test input data and binaries, compiled with and alpha cross-compiler.

```
# select bootscript with --script=this_script_path
m5 checkpoint
htm/st/genome -g256 -s16 -n16384 -t8
m5 exit
```

Figure 8.1: Bootscript to launch genome test with 8 processors, bootscript_htm_genome_8.rcS.

8.2 Modifications and Additions

Several modifications and additions were necessary as part of the merging process of the HTM module within the M5 simulator.

We want the simulator to recognize some new instructions to make possible the execution of the begin and commit transaction instructions among others. For this purpose we will intercept the *XOR* instruction, that called with an specific set of registers will be recognized as a customizable instruction. The Alpha architecture has a register, R_{31}, which is read-only and always contains the value zero. Thus, this register is never used as destination of an operation in correctly compiled code, but, we will hardcode XOR instructions with a specific combination of registers, having R_{31} as destination register. The simulator will detect this kind of combinations after defining them using the special ISA language that M5 uses to define the behavior of the instructions it supports, like we show in Figure 8.2. This way, if the simulator executes an XOR instruction of the form `R31 = R1 ^ R31`, the code in the `htm_begin_tx` function will be executed. Since the R_{31} register is read-only, the execution of this instructions does not affect the processor state at all.

We will use from now on the simulator processor structure instead of our simple FakeCPU, the processor we will use is called *AtomicSimpleCPU*. It is an in-order, one instruction per cycle CPU, that guarantees that once an instruction has started its execution it cannot be interrupted in the middle of it. Many modifications are necessary in the processor side to make our HTM module work properly. First, we have to declare the memory hierarchy to be able to access our cache modules. Then modifications to the *read* and *write* functionalities are necessary to make them use our transactional versions, explained in the previous chapter.

```
0x40: xor({{
  /* For begin_tx: Rc = Ra xor Rb; R31 = R1 xor R31
   * For commit_tx: Rc = Ra xor Rb; R31 = R2 xor R31
   */
  if (this->srcRegIdx(1) == 31 && this->destRegIdx(0) == 31){
    if (this->srcRegIdx(0) == 1){
      fault = xc->htm_begin_tx();
    }
    else if (this->srcRegIdx(0) == 2){
      fault = xc->htm_commit_tx();
    }
    ...
  }
  Rc = Ra ^ Rb_or_imm;
}});
```

Figure 8.2: XOR-based HTM instructions, `src/arch/alpha/isa/decoder.isa`

The processor has a functionality called *tick*, it defines the work done in a per-cycle basis: executing the next instruction and updating the program counter and the system's internal timer, called *current tick*. We modified this functionality, thus at the beginning of this function we check if the HTM is ready, that will either allow or prevent the execution of the next instruction. We prevent the execution of an instruction by launching a fault exception we defined, called *HTMInstrRepeatFault*. It does nothing but prevent the next instruction to be executed. At the end of the function we reschedule this processor to a given current tick, making it wait the time it took the execute the actions taken during this *tick* function execution.

Still in the processor side, we added the register file checkpoint and restore capabilities, used when a begin transaction instruction or an abort occur, respectively. Since we are executing code always in a transactional way, we should be able to support all kind of code. Unfortunately, there are some things like system calls and some specific instructions (e.g., conditional stores) that are not supported inside transactions. The reason is quite obvious, this kind of operations cannot be easily rolled back (if at all). This is a known problem in the TM community and there is specific research to address this problem, there are successful proposals that allow system calls inside transactions with some performance penalties (13). Since it would require a lot of effort to overcome this problem, and it is not part of the project scope either, we will use another solution that we now describe. Even with all the modifications, M5's original memory system is still functional, working in a second plane. Thus, we will switch to it when the system enters in kernel mode, and switch back to transactional execution on returning from kernel mode. This does not affect our results, because going to kernel mode happens in very rear occasions while in the parallel section (part of a test were the threads are running concurrently), and we never enter in kernel mode on explicit transactions. Moreover, the time spent in kernel mode is not counted as part of the execution time in our statistics, so we can thing about this as having cost-free kernel mode executions.

We also implemented our own mechanism to count time spent in the parallel section, by using the simulators current tick variable. So we can easily obtain the time spent inside a region of code by subtracting the exit instant by the entry instant. Finally, we had to adapt the compilation scripts to accept the new source files in the compilation tree.

8.3 Problems Encountered

We found several problems during the whole process of merging the HTM with the simulator. We now discuss the most relevant.

For example, as we explained in the previous section, we are using XOR instructions that have a read-only destination register. The simulator by default was renaming this instructions putting NOPs (No OPeration) instead, because this XOR instructions are useless from the simualtors point of view. It took us some time to notice this fact and we had to disable this feature from the simulator, that was indeed renaming instructions that have R_{31} as destination register.

A very interesting problem we had also is thread migration. The Linux kernel, for some reasons, performs thread migrations between processors. Since with transactions we have all the memory updates in private caches, migrations while a transaction is running must not be allowed, because all the speculatively modified data is lost. Apparently the only solution against a thread migration is to abort the running transaction and restart it in the new processor. But this is not even possible since we will not have the correct register file checkpoint on that processor. To overcome this problem we used the PLPA (Portable Linux

Processor Affinity) library (7). It allows us to associate threads with a particular processor, so that thread will always be executed on that processor, no migrations are allowed. Thus, we had to modify the test sources to pin each thread to a different processor. This solution solved completely the problem.

Chapter 9

Evaluation

In this section we evaluate our HTM module. We first discuss the system configuration. Then, we introduce the methodology used to evaluate our proposal, and finally we present and discuss our results.

9.1 System Configuration

We evaluate our HTM using a Full-System simulator based on Alpha 21264 architecture, that was introduced in the previous chapter. Table 9.1 presents the main parameters of the simulated system.

Feature	**Description**
CPU	1–32 Alpha cores, 2 GHz, in-order, 1 IPC
L1	64-KB, 64-byte cache line 512 direct entries, 2-way associative, 2 cycle latency
L2	1-MB, 64-byte cache line 2K direct entries, 8-way associative, 8 cycle latency
ICN	2D mesh topology, 10 cycles link latency
Main Memory	100 cycles latency

Table 9.1: Parameters for simulated architecture.

The system configuration that we chose tries to be similar to those used in the HTM literature (11; 19; 23; 29; 35), and realistic according to today's hardware possibilities. All operations except loads and stores have a CPI of 1.0. We assume that both L1 and L2 levels of cache are private per processor. When using smaller caches than 1MB we observed poor performance. This is because all the speculative transactional accesses cannot be kept in the cache, resulting in aborts because of the cache evictions. However, nowadays, is common to have 1MB of cache, or even more, at L2 level; and for the benchmarks we evaluated it is enough. Moreover, the HTM module can be extended to support transactions that are larger than private cache sizes by implementing virtualization mechanisms (11; 30).

9.2 Methodology

To evaluate our HTM implementation we will use the *Stanford Transactional Applications for Multi-Processing* (STAMP) benchmarking suite (17). This suite was designed specifically for evaluating TM systems. It consists of eight applications, representative from different application domains; with 30 different sets of configurations and input data that exercise a wide range of transactional behaviors such as short and long transactions, and different sizes of read and write sets, as we will show in the results section (see Section 9.3).

9.2.1 The STAMP Applications

STAMP consists of eight applications: `bayes`, `genome`, `intruder`, `kmeans`, `labyrinth`, `ssca2`, `vacation` and `yada`. Table 9.2 gives a brief description of each benchmark, and a more detailed explanation follows.

1) `bayes` implements an algorithm for learning the structure of Bayesian networks from observed data. A transactional implementation is much simpler than a lock-based approach as using locks would require manually orchestrating a two-phase locking scheme with deadlock detection and recovery to allow concurrent modifications of the graph. `bayes` spends almost all its execution time in long transactions that have large read and write sets. Overall, this benchmark has a high amount of contention as the subgraphs change frequently.

Application	Domain	Description
bayes	machine learning	Learns structure of a Bayesian network
genome	bioinformatics	Performs gene sequencing
intruder	security	Detects network intrusions
kmeans	data mining	Implements K-means clustering
labyrinth	engineering	Routes paths in maze
ssca2	scientific	Creates efficient graph representation
vacation	online transaction processing	Emulates travel reservation system
yada	scientific	Refines a Delaunay mesh

Table 9.2: The applications of the STAMP suite.

2) `genome` implements a process called genome assembly, which is the process of taking a large number of DNA segments and matching them to reconstruct the original source genome. By using transactions a deadlock avoidance scheme is not needed. Overall, the transactions in `genome` are of moderate length and have moderate read and write set sizes. Additionally, almost all of the execution time is transactional, and there is little contention.

3) `intruder` is a signature-based network intrusion detection systems that scans network packets for matches against a known set of intrusion signatures. Overall, this test has a moderate amount of total transactional execution time with moderate to high levels of contention.

4) `kmeans` algorithm groups objects in an N-dimensional space into K clusters. The sizes of the transactions in `kmeans` are relatively small and so are its read and write sets. Overall, this tests spends an small portion of its execution time in transactions.

5) `labyrinth` calculates paths in a three-dimensional grid that represents a maze. Overall, `labyrinth` has very long transactions with very large read and write sets. Virtually all of the code is executed transactionally, and the amount of contention is very high because of the large number of transactional accesses to memory.

6) `ssca2` constructs an efficient graph data structure using adjacency arrays

```
#define BEGIN_TX asm volatile ("xor $1, $31, $31")
#define COMMIT_TX asm volatile ("xor $2, $31, $31")
#define BEGIN_SIMULATION asm volatile ("xor $13, $31, $31")
#define FINISH_SIMULATION asm volatile ("xor $12, $31, $31")
#define SIM_TIME_START asm volatile ("xor $4, $31, $31")
#define SIM_TIME_STOP asm volatile ("xor $5, $31, $31")
#define SIM_PARALLEL_START asm volatile ("xor $6, $31, $31")
#define SIM_PARALLEL_STOP asm volatile ("xor $7, $31, $31")
#define SIM_BARRIER_ENTER asm volatile ("xor $8, $31, $31")
#define SIM_BARRIER_LEAVE asm volatile ("xor $9, $31, $31")
```

Figure 9.1: New directives added to the STAMP tests.

and auxiliary arrays. This application does not spend much time in transactions and the amount of contention is relatively low. Additionally, the length of the transactions and the sizes of their read and write sets is also small.

7) `vacation` implements an online transaction processing system that emulates a travel reservation operations. `vacation` spends a lot of time in transactions and its transactions are of medium length with moderate read and write set sizes. Using transactions simplifies a lot the the parallelization because designing an efficient lock-based strategy is non-trivial.

8) `yada` implements an algorithm for Delaunay mesh refinement. This benchmark has relatively long transactions and spends almost all of its execution time in transactions. Moreover, it has large read and write sets and a moderate amount of contention. This tests was simpler to parallelize, according to its authors, with TM than with locks.

9.2.2 Modifications Required

Some modifications were necessary to adapt the tests to our needs. We added some new directives to execute functionalities present in the simulator side. Figure 9.1 shows the new directives we defined. Note that they use XOR-like instructions similar to the ones we presented in Section 8.2, so when the simulator executes one of this XOR instructions, we intercept it, and we are able to execute a suitable piece of code that implements the functionality we want at that moment.

Using this directives we modified the applications code to define the transactions using our `BEGIN_TX` and `COMMIT_TX`. Likewise, we used the `BEGIN_SIMULATION` directive to instruct the simulator to switch to transactional execution, that is, using our caches and HTM interface, and to start gathering statistics. On the other hand, the `FINISH_SIMULATION` directive switches the simulator back to non-transactional and flushes the statistics. The rest of directives are used to calculate execution times. Thus, `SIM_TIME` directives calculate the time it takes to execute the whole test, while `SIM_PARALLEL` directives calculate the time inside the parallel sections, and `SIM_BARRIER` directives calculate the time spent in barriers for thread synchronization.

To overcome the problem of thread migration we mentioned before, we had to modify the code to associate created threads with a particular processor using the PLPA library. Finally, the compilation of the tests was done using a cross-compiler for Alpha architecture, and the test binaries along with its input data were uploaded to the filesystem of the simulator.

9.3 Results and Discussion

In this section, we analyze the performance of each of the applications to quantify the effectiveness of our HTM module implementation based on the Scalable-TCC protocol (19); we also study the impact of having an always-in-transaction approach and include a detailed overview discussing the results obtained for each application.

All the applications start as single threaded, execute initialization code, and then create worker threads that start executing the parallel part of the application. Since we are interested only in the time spent inside of this parallel section, all the results shown correspond to the parallel section. We used the recommended input configurations and data sets for the tests executions, specified in (17). Moreover, `kmeans` and `vacation` were evaluated with two different conflict rate settings, high and low, as their authors suggest.

Tables 9.3 and 9.4 show a complete set of statistics obtained from executions that range from 1 to 32 processors. While in Figures 9.2 and 9.3 we can see the scalability charts for each test. In the tables we can see the percentage of parallel section time spent in explicit (or real) transactions; the abort rate for both, all (implicit and explicit) transactions, and for implicit transactions; the percentage of parallel section time spent in commit processes for all transactions; the average number of messages sent to a directory per explicit transactional access (excluding messages sent by implicit transactions); percentage of received invalidations that make a transaction (of any kind) abort; average number of wasted read and write accesses per explicit transaction; average number of directories touched per commit for implicit and explicit transactions separately; and the average number of read and writes per explicit transaction.

As we can see in the tables, the time spent in explicit transactions differs significantly from benchmark to benchmark, and explicit transaction sizes range in average from four to over three hundred reads, and from two to over two hundred writes per transaction. This wide range of time spent in transactions and transaction sizes provides a good evaluation of the HTM system.

We can extract several conclusions by looking at the statistics tables. If we take a look at the abort rate columns, we can see that the abort rates considering both implicit and explicit transactions are almost equal compared to explicit transaction abort rates. This means that the abort rate regarding implicit (or forced) transactions is negligible, reaching its maximum in `kmeans-high` with 0.91% (32 processors). This is the first indicator that tells us that the always-in-transaction approach we are using does not introduce sensitive overheads. The second indicator proves this affirmation, the percentage of parallel section time dedicated to commit processes (including both implicit and explicit) is very small, less than 0.14%, for all the tests. We achieve this low overhead at commit time by not writing back data updates during the process, the data remains in private caches until its requested (i.e.,accessed by another processor), and we send messages to the directory to mark and own the lines instead, which are much cheaper than memory updates since we avoid paying the main memory latency for each write done to the same node.

Looking at the number of directory messages sent per transactional access (read or write) in explicit transactions, we can observe that in some tests such as `kmeans-low`, `kmeans-high` or `ssca2`, the number of messages increases substantially while incrementing the number of processors; this is because memory accesses in this tests are using a lot of shared data, which requires a higher amount of communication between nodes. However, in the worst case, the number of messages increases linearly with the number of processors; and in the common case the increase is very small, compensating the fact that main memory is being split over more nodes.

One of the key ideas that makes the protocol scale is the possibility of having parallel commits of transactions that touch disjoint sets of directories. Thus, for us is very important to know the average number of directories touched per commit in both implicit and explicit transactions. As shown in the table, for explicit transactions, the number of directories touched per commit is small in the common case. With a maximum of five directories touched in the wost case, and an average of 2.73 directories touched for 8 processors and 3.26 for 32 processors. This numbers allow for a very good potential degree of parallel commits. On the other side, for implicit transactions, the average number of directories touched is very small, being lower than one in many cases, which suggest a large number of read-only implicit transactions.

We see from the speedup curves in Figures 9.2 and 9.3 that the overall performance of our HTM implementation is good compared to existing implementations. Average speedup with 16 processors is 8.09 and for 32 processors is 12.86. We noted that some applications, `genome` and `ssca2`, have difficulties to scale because of wide use of barriers for synchronizing the execution. The time spent in barriers for these applications is shown in the figures with a green line and triangle markers. As we can see, barriers severely damage scalability since the time spent inside them increase considerably with the number of processors.

Looking at the applications in more detail gives us more insight into the behavior of the HTM module.

1) `bayes` has a discreet speedup for a low number of processors but has an unexpected performance boost when executing with more than 8 processors. Since bayes has large transactions, as we can see in Table 9.4, we thought this boost of performance was because of a sudden increase of hits in one of the cache levels, due to hot memory addresses fitting in caches. However, we checked this possibility by looking at the cache statistics, and we found that the number of hits in all the memory hierarchy levels remain quite constant comparing the executions

Application	Proc	%ETX	%ABO TOT	%ABO ETX	%CMT TOT	#MSGs TX ACS	%INVAL ABO	#WRD ETX	#WWR ETX	#DIRs ETX	#DIRs ITX	#RD ETX	#WR ETX
ssca2	1	16.6	0.00	0.00	0.003	1.00	0.00	0.00	0.00	1.00	0.24	4.00	2.00
	2	20.7	0.00	0.00	0.010	1.33	58.82	0.00	0.00	1.50	0.20	4.00	2.00
	4	18.6	0.01	0.01	0.020	1.78	68.37	0.01	0.00	1.71	0.24	4.00	2.00
	8	15.6	0.04	0.04	0.035	2.56	67.62	0.01	0.00	1.88	0.28	4.00	2.00
	16	10.9	0.06	0.06	0.062	3.93	63.03	0.02	0.01	1.94	0.26	4.00	2.00
	32	5.5	0.16	0.16	0.069	6.63	58.45	0.04	0.02	1.98	0.28	4.00	2.00
intruder	1	50.7	0.00	0.00	0.007	0.62	0.00	0.00	0.00	1.00	0.12	16.22	5.51
	2	56.7	1.60	1.57	0.015	0.72	13.35	1.05	0.27	1.33	0.18	16.88	5.51
	4	65.4	6.00	5.95	0.026	0.89	23.83	5.00	1.32	2.22	0.21	16.84	5.51
	8	77.0	15.31	15.27	0.039	1.12	36.01	16.36	4.25	2.63	0.23	16.72	5.50
	16	85.3	31.62	31.59	0.052	1.45	49.75	39.73	9.78	2.77	0.26	16.92	5.49
	32	90.2	50.58	50.56	0.065	2.10	60.11	81.75	20.74	3.02	0.27	16.97	5.48
kmeans-high	1	13.0	0.00	0.00	0.002	0.67	0.00	0.00	0.00	1.00	0.09	6.50	1.75
	2	15.9	0.21	0.09	0.005	0.88	56.84	0.04	0.01	1.00	0.09	6.50	1.75
	4	17.4	1.01	0.44	0.011	1.21	81.69	0.23	0.04	1.00	0.09	6.49	1.75
	8	19.2	2.44	1.32	0.027	1.81	82.74	0.71	0.12	1.00	0.09	6.49	1.75
	16	24.6	5.76	4.48	0.065	2.91	71.48	2.63	0.50	1.03	0.09	6.47	1.75
	32	46.7	23.01	22.10	0.104	4.87	64.73	16.87	3.44	1.00	0.09	6.45	1.74
kmeans-low	1	5.5	0.00	0.00	0.001	0.67	0.00	0.00	0.00	1.00	0.04	6.50	1.75
	2	6.8	0.04	0.01	0.002	0.87	93.18	0.01	0.00	1.00	0.04	6.50	1.75
	4	7.5	0.21	0.03	0.005	1.20	96.36	0.03	0.00	1.00	0.04	6.49	1.75
	8	7.9	0.40	0.10	0.013	1.80	80.66	0.11	0.02	1.01	0.04	6.49	1.75
	16	8.4	0.69	0.29	0.032	2.85	86.33	0.30	0.05	1.00	0.04	6.47	1.75
	32	9.4	1.24	0.76	0.078	4.99	83.58	0.77	0.11	1.01	0.04	6.45	1.74
labyrinth	1	99.8	0.00	0.00	0.001	0.81	0.00	0.00	0.00	1.00	0.85	315.53	212.06
	2	99.7	10.15	10.15	0.002	0.80	5.58	185.59	118.73	1.99	1.12	304.47	208.18
	4	100.0	22.29	22.29	0.002	0.80	6.31	592.56	382.32	2.96	1.26	308.89	204.16
	8	99.3	32.20	32.2	0.002	0.79	6.55	1390.48	874.65	3.83	1.24	287.32	196.36
	16	99.6	39.21	39.21	0.002	0.80	6.92	2765.76	1746.32	4.79	1.19	275.75	182.19
	32	100.0	46.34	46.34	0.002	0.85	9.25	5202.16	3550.37	5.26	1.20	235.34	160.00

Table 9.3: HTM Execution Statistics (1/2).

Legend: **%ETX** — Percentage of parallel section time spent inside explicit transactions; **%ABO** — Percentage of aborts (abort rate), calculated as aborts/(aborts+commits_total); **%CMT** — Percentage of parallel section time dedicated to commit processes; **#MSGs TX ACS** — Average number of messages sent to a directory per explicit transactional access; **%INVAL ABO** — Percentage of received invalidations that make a transaction (implicit or explicit) abort; **#WRD/#WWR ETX** — Average number of wasted read/write accesses per explicit transaction; **#DIRs ETX/ITX** — Average number of directories touched per commit in explicit/implicit transactions; **#RD/#WR ETX** — Average number of read/writes per explicit transaction.

Application	Proc	%ETX	%ABO TOT	%ABO ETX	%CMT TOT	#MSGs TX ACS	%INVAL ABO	#WRD ETX	#WWR ETX	#DIRs ETX	#DIRs ITX	#RD ETX	#WR ETX
genome	1	93.3	0.00	0.00	0.005	0.32	0.00	0.00	0.00	0.96	0.31	35.32	4.80
	2	89.2	0.26	0.25	0.011	0.38	3.49	0.28	0.05	1.66	0.26	35.42	4.75
	4	75.4	0.76	0.76	0.021	0.49	5.58	0.66	0.11	2.28	0.27	35.21	4.73
	8	55.8	1.59	1.52	0.034	0.67	7.59	1.50	0.26	2.72	0.28	35.32	4.73
	16	28.2	3.61	3.56	0.047	0.97	13.62	3.66	0.54	3.01	0.29	35.42	4.72
	32	10.2	5.46	5.45	0.027	1.48	16.53	5.54	0.85	3.17	0.27	35.37	4.73
yada	1	100.0	0.00	0.00	0.003	0.61	0.00	0.00	0.00	0.72	0.34	50.76	19.06
	2	100.0	4.72	4.72	0.005	0.68	4.87	11.46	4.14	1.25	0.47	49.56	18.19
	4	99.6	9.70	9.70	0.008	0.74	5.61	24.00	8.71	1.58	0.67	49.67	17.99
	8	100.0	15.42	15.42	0.014	0.81	6.76	40.21	14.65	2.05	1.19	49.31	17.53
	16	100.0	22.45	22.45	0.027	0.90	9.06	55.82	20.36	2.59	2.07	49.07	17.56
	32	89.3	34.29	34.29	0.037	1.03	12.11	87.95	31.12	2.75	2.21	48.42	17.16
vacation-high	1	48.3	0.00	0.00	0.002	0.34	0.00	0.00	0.00	1.00	0.49	90.27	11.80
	2	48.9	0.00	0.00	0.005	0.37	0.07	0.01	0.00	1.98	0.58	91.12	11.13
	4	53.9	0.00	0.00	0.011	0.43	0.27	0.18	0.01	3.25	0.72	90.90	10.75
	8	60.0	0.04	0.04	0.025	0.54	0.53	0.43	0.03	4.27	1.12	91.01	10.55
	16	62.6	0.15	0.15	0.054	0.73	0.67	0.72	0.04	4.94	1.91	91.46	10.49
	32	66.5	0.71	0.70	0.119	1.02	2.10	2.51	0.15	5.28	2.22	90.98	10.44
vacation-low	1	50.9	0.00	0.00	0.003	0.38	0.00	0.00	0.00	1.00	0.45	76.61	11.18
	2	48.8	0.00	0.00	0.005	0.40	0.15	0.01	0.00	2.00	0.48	77.02	10.46
	4	56.5	0.00	0.00	0.013	0.47	0.07	0.02	0.00	3.23	0.89	76.57	10.09
	8	63.4	0.06	0.06	0.031	0.60	0.72	0.25	0.02	4.11	1.51	76.82	9.96
	16	64.9	0.17	0.17	0.067	0.80	1.05	0.48	0.03	4.67	2.04	76.41	9.83
	32	63.2	0.29	0.28	0.139	1.11	1.57	0.69	0.05	4.96	2.12	76.57	9.79
bayes	1	57.4	0.00	0.00	0.001	0.64	0.00	0.00	0.00	1.00	0.99	135.55	59.69
	2	64.2	0.15	0.15	0.002	0.67	23.66	45.23	23.17	1.69	1.24	145.10	66.10
	4	66.2	0.29	0.29	0.003	0.69	17.55	62.56	28.33	3.01	1.37	140.35	62.03
	8	90.4	1.58	1.57	0.005	0.73	45.95	288.88	128.06	3.77	1.43	145.60	68.04
	16	100.0	1.18	1.15	0.017	0.77	37.23	210.59	102.53	4.61	1.47	155.41	78.06
	32	100.0	1.25	1.22	0.035	0.84	43.48	198.47	95.26	4.16	1.48	138.40	66.63

Table 9.4: HTM Execution Statistics (2/2).

Legend: **%ETX** — Percentage of parallel section time spent inside explicit transactions; **%ABO** — Percentage of aborts (abort rate), calculated as aborts/(aborts+commits_total); **%CMT** — Percentage of parallel section time dedicated to commit processes; **#MSGs TX ACS** — Average number of messages sent to a directory per explicit transactional access; **%INVAL ABO** — Percentage of received invalidations that make a transaction (implicit or explicit) abort; **#WRD/#WWR ETX** — Average number of wasted read/write accesses per explicit transaction; **#DIRs ETX/ITX** — Average number of directories touched per commit in explicit/implicit transactions; **#RD/#WR ETX** — Average number of read/writes per explicit transaction.

with 8 and 32 processors. Therefore, after inspecting the source code and further examining the STAMP paper (17) we have found that the execution time is sensitive to the order in which the dependencies of the Bayesian network are learned. Thus, the speedup curves are not as smooth as those for other STAMP applications.

2) `genome` scales constantly up to 8 processor configurations (see Figure 9.3). However, the barriers used to synchronize the execution saturate scalability and make the application lose performance with 16 and 32 processors, were the processors spend, respectively, 47% and 59% of the parallel section execution time in barriers. We can also see the effect of barriers on the percentage of parallel section time spent inside explicit transactions, shown in the first column of Table 9.4.

3) `intruder` curve shows a constant speedup, this is because the application suffers from high abort rates, over 50% with 32 processors which severely degrades scalability. This high abort rates make wasted memory access rise a lot, having more than one hundred wasted accesses per explicit transaction. However, we already expected the curve to be this way since `intruder` has a high conflict rate.

4) `kmeans` is evaluated with two different inputs resulting in low and high contention (see Figure 9.2). For low level of contention it scales very well, up to 21.5x with 32 processors. On the other hand, for high level of contention, it scales at the same rate up to 8 processors, but with more processors the abort rate starts to be significant reaching a 23%. Thus, the obtained speedup for 32 processors is lower than the one we obtained for low contention.

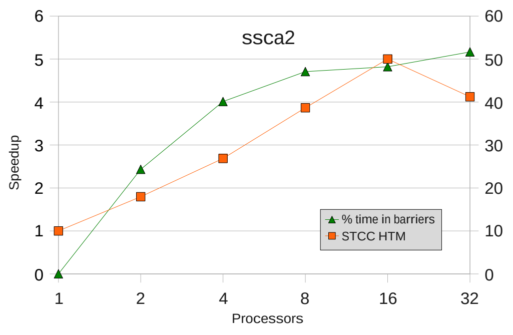
5) `labyrinth` is the test with the worst results. The reason is a high abort rate combined with long running transactions, over five hundred memory accesses per explicit transaction. This combination generates a lot of wasted work, as we can see in Table 9.3, with almost nine thousand wasted memory accesses per explicit transaction. However, we have found, after looking at the STAMP paper (17), that for `labyrinth` to scale an *early release* (32; 33) of cache lines has to be supported by the HTM. Early release means that a cache line is explicitly removed from the transaction during the transaction execution, that it, before it commits. In `labyrinth's` application main loop, every transaction reads an entire dataset, and it consequently becomes part of the read set. On commit every transaction wants to mark as owned some part of the dataset, so this part of the execution is guaranteed to create conflicts on every commit. Since we do not support early release, we have a fully conflicting execution.

6) `ssca2` has, like `genome`, barriers to synchronize the execution. In this case, we achieve good scalability up to 16 processors while having around 50% of the parallel execution time spend in barriers (Figure 9.2). This test has very small transactions and a very low abort rate as well (see Table 9.3). Thus, speedups are drastically constrained because of the barriers, which already consume 25% of the execution time with 2 processors.

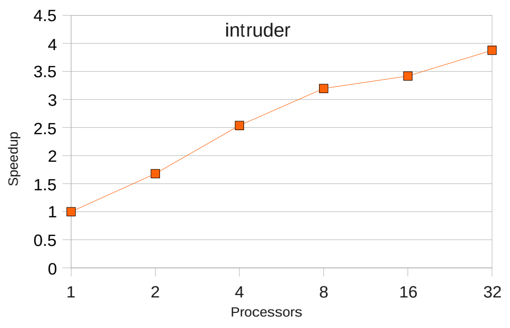
7) `vacation` is the best performing application, and is evaluated with two different conflict rates, high and low. As we can see in Figure 9.3 we have almost the same speedups for both conflict rates, around 15x for 16 processors and 27x for 32 processors, presenting a very good scalability for all the configurations. In this case, a higher amount of contention is not making the abort rate increase, thus performance is not damaged as happened in `kmeans-high` with high processor counts.

8) `yada` has a significant abort rate even with low processor counts (see Table 9.4), and even though the abort rate increases up to 34% with 32 processors, it does so in a constant rate. This gives the application some margin to scale in the same way for all the configurations, reaching a speedup above 8x for 32 processors.

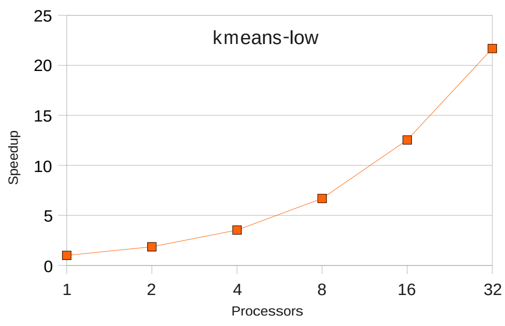
In general, all the speedup curves obtained match the expected results according to the characteristics of each application, and, in most cases, there are significant speedups. We have shown that having an always-in-transaction approach does not damage system performance nor introduces significant overheads, while leads to a simpler hardware design (non-transactional state tracking is not needed).



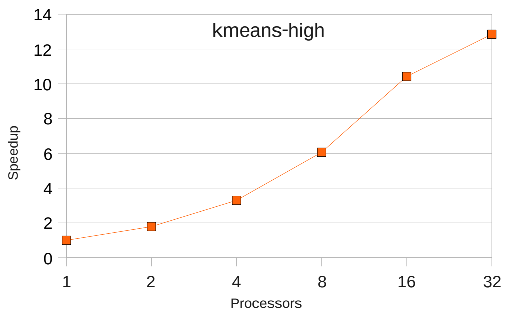
(a) ssca2



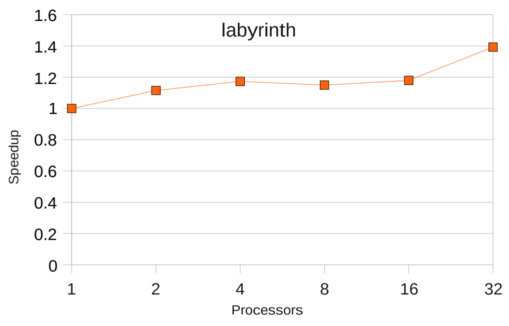
(b) intruder



(c) kmeans-low

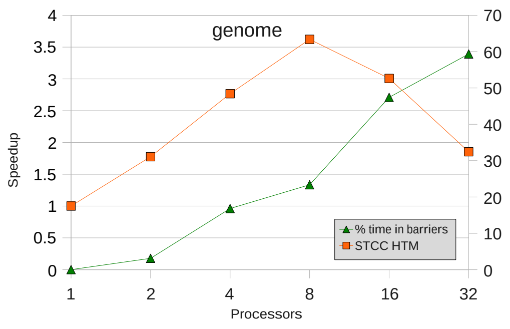


(d) kmeans-high

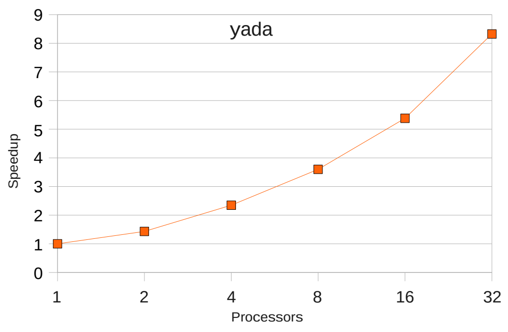


(e) labyrinth

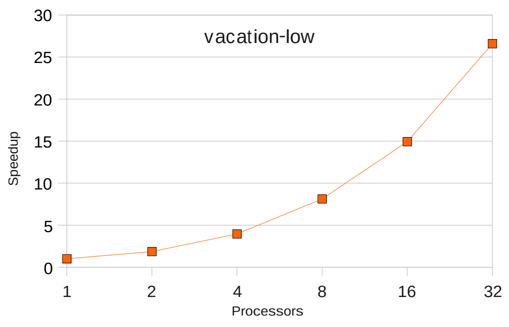
Figure 9.2: Scalable TCC HTM execution time, parallel section speedup over single-threaded execution (1/2).



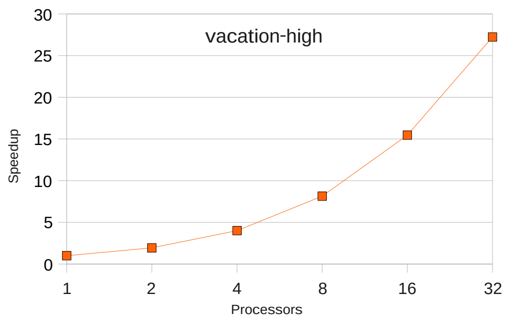
(a) genome



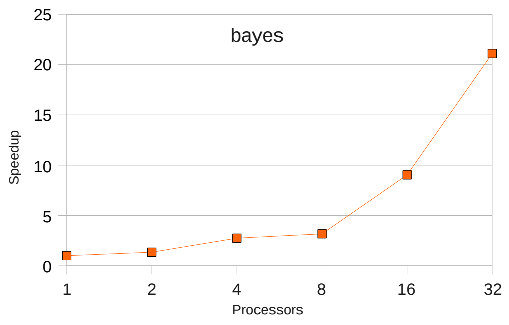
(b) yada



(c) vacation-low



(d) vacation-high



(e) bayes

Figure 9.3: Scalable TCC HTM execution time, parallel section speedup over single-threaded execution (2/2).

Chapter 10

Project Analysis

This chapter exposes the project costs and its initial and final project time planning. It also discuss the achieved objectives and future lines of work. Finally, personal conclusions conclude the chapter.

10.1 Project Planning

The planning of this project was very hard to do when the project started, since the topic of the project, transactional memory, was completely unknown to us, and we had no clue about how much time would take to learn the necessary knowledge to achieve our objectives.

The project workload is split into three different work packages:

1. Introduction and generic modifications to The M5 Simulator.
2. Scalable-TCC HTM module.
3. Merge Scalable-TCC HTM module with The M5 Simulator.

For the reasons explained above, the initial plan was conceived after starting the second work package, because until than moment we had no background to know how many time would require to accomplish each task, and some tasks were still not clear.

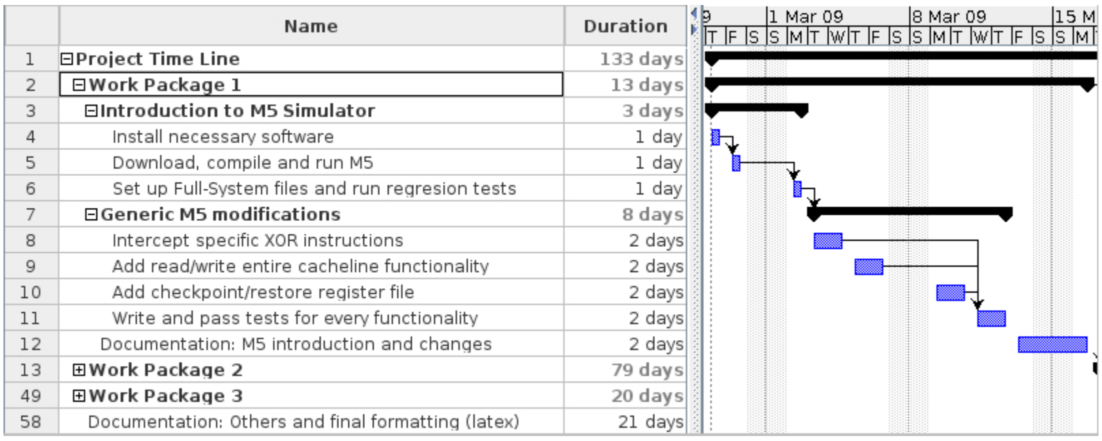
The project started the 26th of February and had an approximate duration of 6 months, ∼26 weeks, having as deadline August 31st. Fortunately, the project objectives were well defined since the beginning; fact that allowed us, after having some background of the project's topic and a general view of the tasks necessary to complete each work package, to define an accurate planning. Thus, delaying the creation of the initial plan a few weeks helped a lot to maintain the plan almost unchanged in terms of tasks, and time estimations were also more accurate since we had some hints about how much time would require each task to complete.

We now present the initial plan of time estimations for each work package, with its tasks detailed; followed by the modifications required after each package was completed.

10.1.1 Work Package 1

As we explained in the previous section, this work package was finished before creating the initial plan. However, this package has a very small amount of workload compared to the other ones, it took 13 days to complete.

In this package we learned about The M5 Simulator structure and its tools. Moreover, after inspecting and get familiar with M5's source code, some generic modifications have been done to the simulator that will be useful in the future when merging the Scalable-TCC HTM module into it.



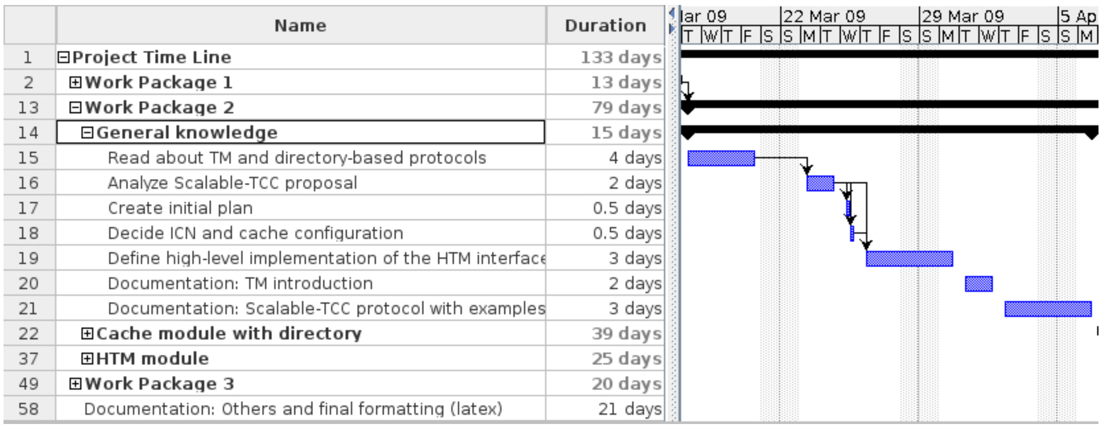
	Name	Duration
1	⊟Project Time Line	133 days
2	⊟Work Package 1	13 days
3	⊟Introduction to M5 Simulator	3 days
4	Install necessary software	1 day
5	Download, compile and run M5	1 day
6	Set up Full-System files and run regresion tests	1 day
7	⊟Generic M5 modifications	8 days
8	Intercept specific XOR instructions	2 days
9	Add read/write entire cacheline functionality	2 days
10	Add checkpoint/restore register file	2 days
11	Write and pass tests for every functionality	2 days
12	Documentation: M5 introduction and changes	2 days
13	⊞Work Package 2	79 days
49	⊞Work Package 3	20 days
58	Documentation: Others and final formatting (latex)	21 days

Figure 10.1: Work package 1 Gantt diagram, initial plan.

Figure 10.1 shows the Gantt diagram for this package. The time assigned to each task is not estimated, since the diagram was done after the package was complete, so no modifications apply.

10.1.2 Work Package 2

This work package has the largest volume of work. Thus, the package itself is divided in three sub packages. During the first one we read extensively about our topic of research and we also developed the initial plan among other tasks shown in Figure 10.2. The second packet focuses on the cache with directory module (see Figure 10.3); and the third, on the HTM module (see Figure 10.4).

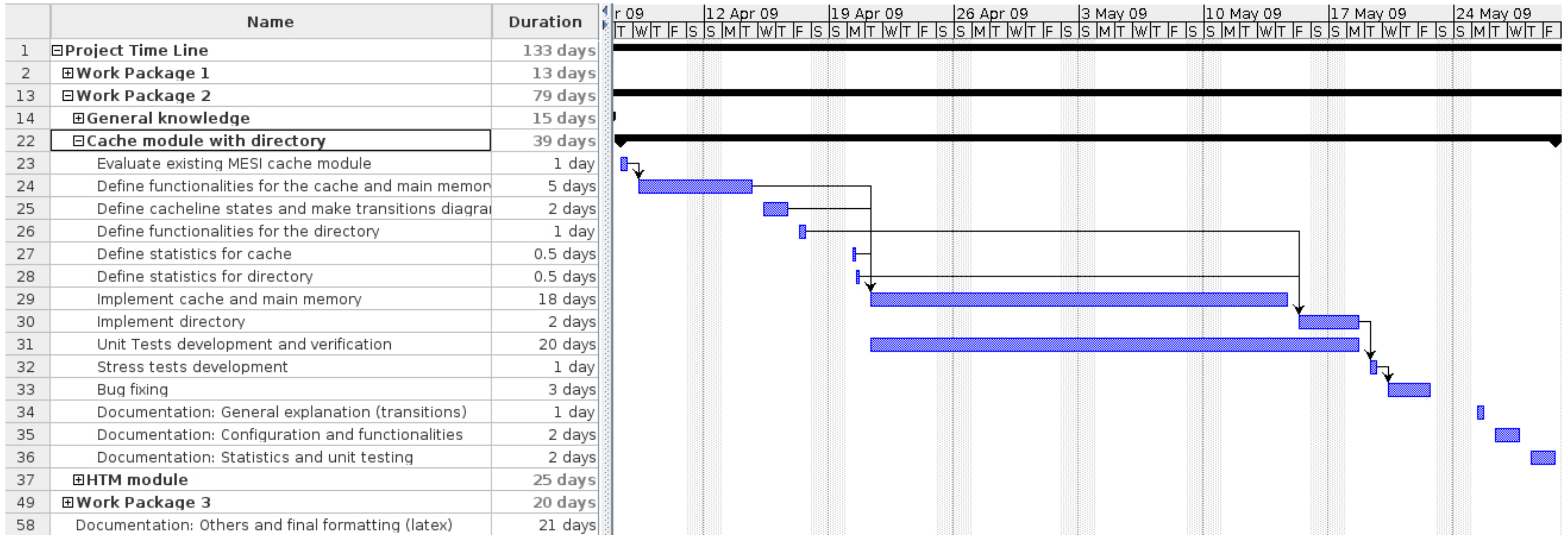


	Name	Duration
1	⊟Project Time Line	133 days
2	⊞Work Package 1	13 days
13	⊟Work Package 2	79 days
14	⊟General knowledge	15 days
15	Read about TM and directory-based protocols	4 days
16	Analyze Scalable-TCC proposal	2 days
17	Create initial plan	0.5 days
18	Decide ICN and cache configuration	0.5 days
19	Define high-level implementation of the HTM interface	3 days
20	Documentation: TM introduction	2 days
21	Documentation: Scalable-TCC protocol with examples	3 days
22	⊞Cache module with directory	39 days
37	⊞HTM module	25 days
49	⊞Work Package 3	20 days
58	Documentation: Others and final formatting (latex)	21 days

Figure 10.2: Work package 2 Gantt diagram, first sub package, initial plan.

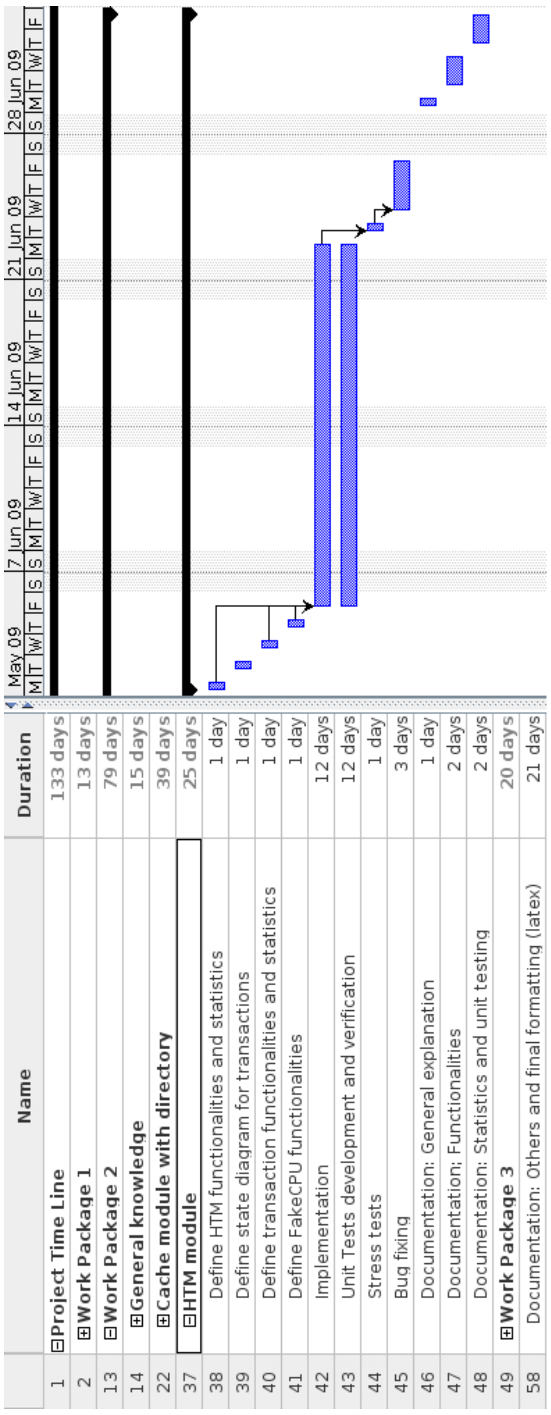
The tasks for this first sub package were very straightforward and there was no problem to accomplish all the tasks on time. Note that the time of the first two task is not an estimation, since the initial plan was created after reading and analyzing the Scalable-TCC proposal. Actually, we saved one day in the high-level definition of the HTM interface, so this sub package was completed one day before its scheduled deadline.

In the second sub package (see Figure 10.3), that started one day earlier than the Gantt diagram shows, we had some time miss prediction in the implementation task, since combining implementation with unit testing reveals some bugs that need to be fixed at the moment, and the implementation time increases; it took five extra days. In contrast, bug fixing task was less time consuming since



	Name	Duration
1	⊟Project Time Line	133 days
2	⊞Work Package 1	13 days
13	⊟Work Package 2	79 days
14	⊞General knowledge	15 days
22	⊟Cache module with directory	39 days
23	Evaluate existing MESI cache module	1 day
24	Define functionalities for the cache and main memor	5 days
25	Define cacheline states and make transitions diagrai	2 days
26	Define functionalities for the directory	1 day
27	Define statistics for cache	0.5 days
28	Define statistics for directory	0.5 days
29	Implement cache and main memory	18 days
30	Implement directory	2 days
31	Unit Tests development and verification	20 days
32	Stress tests development	1 day
33	Bug fixing	3 days
34	Documentation: General explanation (transitions)	1 day
35	Documentation: Configuration and functionalities	2 days
36	Documentation: Statistics and unit testing	2 days
37	⊞HTM module	25 days
49	⊞Work Package 3	20 days
58	Documentation: Others and final formatting (latex)	21 days

Figure 10.3: Work package 2 Gantt diagram, second sub package, initial plan.



	Name	Duration
1	⊟**Project Time Line**	133 days
2	⊞**Work Package 1**	13 days
13	⊟**Work Package 2**	79 days
14	⊞**General knowledge**	15 days
22	⊞**Cache module with directory**	39 days
37	⊟**HTM module**	25 days
38	Define HTM functionalities and statistics	1 day
39	Define state diagram for transactions	1 day
40	Define transaction functionalities and statistics	1 day
41	Define FakeCPU functionalities	1 day
42	Implementation	12 days
43	Unit Tests development and verification	12 days
44	Stress tests	1 day
45	Bug fixing	3 days
46	Documentation: General explanation	1 day
47	Documentation: Functionalities	2 days
48	Documentation: Statistics and unit testing	2 days
49	⊞**Work Package 3**	20 days
58	Documentation: Others and final formatting (latex)	21 days

Figure 10.4: Work package 2 Gantt diagram, third sub package, initial plan.

the major part of the bugs were identified during the implementation, it took one day. The rest of the tasks were completed in the time predicted. Overall, after finishing this sub package we were two days delayed from initial schedule.

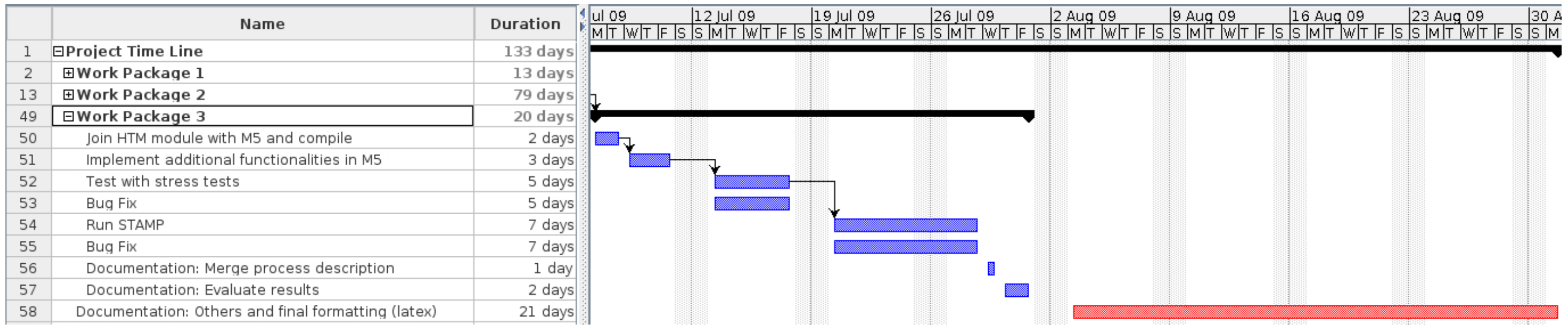
Thus, the last sub package starts with two days of delay. As happened with the previous one we had the same problem with the implementation task, it took three extra days, and we saved one day in the bug fixing task. Having unit tests helped a lot to find difficult bugs and to maintain functionalities working during the implementation process, even though it took some extra time. Thus, this sub package and the whole second work package finished with four days of delay.

10.1.3 Work Package 3

The last work package consists in the merge of the HTM module with the simulator and results evaluation. We include here for completion the last task of the project, documentation and formatting. Figure 10.5 shows the initial Gantt diagram for this package.

We underestimated the time to run successfully the stress tests at this final stage, some hard to track bugs came up and resulted to be time consuming. Thus, the task took three extra days; in contrast, the time it took to run successfully the STAMP tests, since we already fixed a lot of bugs in the stress test executions, took one day less. However, when we did the initial planning we forgot an important task, the modifications required in the STAMP tests source code, task that took 2 days to accomplish.

Overall, when we finished the tasks of the third package, excluding the final documentation task, we had eight days of delay. However, at that time more or less, we were able to move the deadline of the project to the 9^{th} of September, giving us seven extra days to finish the project. Even with this extra time, we had to work hard to finish this documentation in time, since writing in a foreign language is much more difficult than in a native language.



	Name	Duration
1	⊟Project Time Line	133 days
2	⊞Work Package 1	13 days
13	⊞Work Package 2	79 days
49	⊟Work Package 3	20 days
50	Join HTM module with M5 and compile	2 days
51	Implement additional functionalities in M5	3 days
52	Test with stress tests	5 days
53	Bug Fix	5 days
54	Run STAMP	7 days
55	Bug Fix	7 days
56	Documentation: Merge process description	1 day
57	Documentation: Evaluate results	2 days
58	Documentation: Others and final formatting (latex)	21 days

Figure 10.5: Work package 3 Gantt diagram, initial plan.

10.2 Project Cost

In this section we present the economic study of the project costs. The costs of the project are divided in human, hardware and software costs.

10.2.1 Human Resources

This project has been developed in a research center which targets mainly PhD students. This fact conditions this study since the organization in this kind of centers differs from normal companies organizations.

The development of this project required the intervention of an analyst, a developer, and an advisor or project manager. Table 10.1 shows the human costs separated per working role, hours of dedication and estimated salary per hour.

From the Gantt diagram we can see that the initial plan included 133 days of work, but after changing the deadline of the project, it took a total of 140 days to complete, 1120 hours.

Role	**Hours of dedication**	**Price/Hour**	**Total**
Analyst	450h	25€	11250€
Developer	670h	20€	13400€
Project manager	30h	35€	1050€
Total			**25700€**

Table 10.1: Total human resources costs.

10.2.2 Hardware Resources

Regarding hardware costs, we have to consider a personal computer and a server where the test were executed. Moreover, an additional screen was also provided since working on laptop screen is not comfortable. Table 10.2 summarizes the costs.

Hardware	**Description**	**Cost**
Personal Computer	Lenovo Thinkpad T400	1350€
Monitor	Fujitsu ScenicView B19-2	125€
Server	4 x Intel Xeon 5160 3GHz; 16GB RAM	2750€
Total		**4225€**

Table 10.2: Total hardware resources costs.

10.2.3 Software Resources

All the software used to develop this project has free license, starting from the Ubuntu Linux operating system. We now enlist all the applications used during the project:

- *Mercurial:* is a distributed version control system.
- *Doxygen:* is a documentation system, it can generate an on-line documentation browser and an off-line reference manual from a set of documented source files.
- *CMake:* is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner.
- *Vim:* is an advanced text editor that seeks to provide the power of the de-facto Unix editor 'Vi', with a more complete feature set.
- *QuickTest:* is a simple C++ unit testing framework.
- *OpenProj:* is a free, open source desktop alternative to Microsoft Project.
- *Kile:* is a user friendly TEX/LATEXeditor.
- *LATEX:* is a document markup language and document preparation system for the TEXtypesetting program.
- *The M5 Simulator:* is a modular platform for computer system architecture research.

- *Scons:* is a software construction tool (build tool, or make tool) implemented in Python.
- *Python:* is a programming language that lets you work more quickly and integrate your systems more effectively.
- *OpenOffice:* is a free office applications suite.
- *Inkscape:* is an open source vector graphics editor, with a lot of capabilities.

10.2.4 Total Cost

Table 10.3 shows the total cost of the project, considering: human, hardware and software costs.

Resources	**Cost**
Human	25700€
Hardware	4225€
Software	0€
Total	**29925€**

Table 10.3: Total costs.

10.3 Achieved Objectives

At the end of this project both main objectives have been accomplished.

We have successfully developed an HTM simulator. We obtained, through full-system simulation, significant speedups that reach 27x for configurations with 32 processors, making TM a promising approach for parallel programming. Moreover, our HTM implementation allows for multiple configuration options. This options can be tuned to user needs to see its effect in the results, where our complete statistics system provides a lot of information about the executions.

Furthermore, our results are used to compare performance with another approach developed in the Center that is presented in an upcoming article that will be published in a near future (35).

10.4 Future Work

Even though the simulator is fully functional now, there are a lot of things to work on to make it more complete.

One of these extra functionalities would be additional support for long-running transactions that overflow hardware capabilities, using one of the approaches we already commented.

It would also be interesting to provide a user-friendly way to modify the configuration of the simulator, since in the actual version it is possible only by directly modifying the source code definitions. Thus, using a text configuration file, for example, would make much easier to change the configuration and to automate the process of running the simulator with different configurations.

Finally, we would like to add support for nested transactions, since there are applications that use them (not in STAMP) and we would like to be able to execute as much applications as possible.

10.5 Personal Conclusions

During these months in the Barcelona Supercomputing Center – Microsoft Research I have learned a lot of things, in both technical and personal sides.

The project was a challenge from the beginning, since the topic was completely unknown to me. I had to learn all the necessary concepts fast to understand clearly the proposals I had to analyze. Moreover, working with the simulator was difficult at some stages, for example, the final executions were very difficult to debug because the simulator is very complex. This experiences helped me to mature in a professional way since I had to face many problems on my own and to take decisions based in my own criteria.

I have had the opportunity to make this project in an international environment, with researchers of several countries from all over the world, that have helped and taught me a lot of things. From their experience and endless skills I have learned different work methodologies and new ways to do things much more efficiently. This environment helped me to focus and pushed me to keep learning every day.

Bibliography

[1] BSC - Microsoft Research Center – www.bscmsrc.eu. 1

[2] Cache – http://en.wikipedia.org/wiki/CPU_cache. 30

[3] Doxygen – http://www.stack.nl/ dimitri/doxygen/index.html. 40

[4] M5 publications list – http://www.m5sim.org/wiki/index.php/Publications. 23

[5] The M5 Simulator site – http://www.m5sim.org/wiki/index.php/Main_Page. 63

[6] Moore's law – http://en.wikipedia.org/wiki/Moore%27s_law. 1

[7] Plpa site – http://www.open-mpi.org/projects/plpa/. 67

[8] Problems with locks – http://en.wikipedia.org/wiki/Lock_(computer_science). 3

[9] Quicktest – http://quicktest.sourceforge.net/. 44

[10] Test-driven development – http://en.wikipedia.org/wiki/Test-driven_development. 43

[11] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, San Francisco, California, February 2005. 26, 68, 69

[12] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, **26**(4):52–60, 2006. 5, 23

[13] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr 2006. 65

[14] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. *SIGARCH Comput. Archit. News*, **35**(2):81–91, 2007. 12, 13

[15] Shekhar Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, **19**(4), 1999. 1

[16] Broadcom. The Broadcom BCM1250 multiprocessor. April 2002. 4

[17] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008. 7, 69, 72, 77

[18] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007. 14

[19] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *13th International Symposium on High Performance Computer Architecture (HPCA)*. Feb 2007. 5, 10, 15, 19, 26, 68, 72

[20] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in Transactional Memory Virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM Press, Oct 2006. 30

[21] JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, Christos Kozyrakis, , and Kunle Olukotun. The common Case Transactional Behavior of Multithreaded Programs. In *12th International Symposium on High Performance Computer Architecture (HPCA)*. Feb 2006. 30

[22] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with Transactional Coherence and Consistency (TCC). In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13. ACM Press, Oct 2004. 3

[23] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004. 26, 68

[24] Tim Harris, Adrián Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. Transactional Memory: An Overview. *IEEE Micro*, **27**(3):8–29, 2007. 10, 11, 13

[25] Maurice Herlihy, J. Eliot B. Moss, J. Eliot, and B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993. 2, 3

[26] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way Multithreaded Sparc Processor. *IEEE Micro*, **25**(2):21–29, 2005. 4

[27] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2006. 11

[28] Victor Luchangco. Beyond Simple Transactions and Atomic Blocks. 9

[29] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Log-TM: Log-based Transactional Memory. In *in HPCA*, pages 254–265, 2006. 26, 68

[30] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. *International Symposium on Computer Architecture*, **0**:494–505, 2005. 4, 26, 30, 69

[31] Arrvindh Shriraman and Sandhya Dwarkadas. Refereeing conflicts in hardware transactional memory. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 136–146, New York, NY, USA, 2009. ACM. 13

[32] T. Skare and C. Kozyrakis. Early release: Friend or foe. In *Workshop on Transactional Memory Workloads*, 2006. 77

[33] N. Sonmez, C. Perfumo, S. Stipic, A. Cristal, O.S. Unsal, and M. Valero. unreadTVar: Extending haskell software transactional memory for performance. In *Proc. of Eighth Symposium on Trends in Functional Programming (TFP 2007)*, 2007. 77

[34] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 141–150, New York, NY, USA, 2009. ACM. 12

[35] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. EazyHTM, Eager-Lazy Hardware Transactional Memory. In *(To appear) 42nd International Symposium on Microarchitecture, Micro'09*. Dec 2009. 4, 14, 27, 68, 90

[36] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *In HPCA 13*, pages 261–272, 2007. 24