DATA CENTRIC AND ADAPTIVE SOURCE CHANGING TRANSACTIONAL MEMORY WITH EXIT FUNCTIONALITY

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2012

By
Herath Mudiyanselage Isuru Prasenajith Herath
School of Computer Science

Contents

Word Count: 57081

List of Tables

List of Figures

Abstract

DATA CENTRIC AND ADAPTIVE SOURCE CHANGING
TRANSACTIONAL MEMORY WITH EXIT FUNCTIONALITY
Herath Mudiyanselage Isuru Prasenajith Herath
A thesis submitted to The University of Manchester
for the degree of Doctor of Philosophy, 5th December 2012

Multi-core computing is becoming ubiquitous due to the scaling limitations of single-core computing. It is inevitable that parallel programming will become the mainstream for such processors. In this paradigm shift, the concept of abstraction should not be compromised. A programming model serves as an abstraction of how programs are executed. Transactional Memory (TM) is a technique proposed to maintain lock free synchronization. Due to the simplicity of the abstraction provided by it, TM can also be used as a way of distributing parallel work, maintaining coherence and consistency. Motivated by this, at a higher level, the thesis makes three contributions and all are centred around Hardware Transactional Memory (HTM).

As the first contribution, a transaction-only architecture is coupled with a "data centric" approach, to address the scalability issues of the former whilst maintaining its simplicity. This is achieved by grouping together memory locations having similar access patterns and maintaining coherence and consistency according to the group each memory location belongs to. As the second contribution a novel technique is proposed to reduce the number of false transaction aborts which occur in a signature based HTM. The idea is to adaptively switch between cache lines and signatures to detect conflicts. That is, when a transaction fits in the L1 cache, cache line information is used to detect conflicts and signatures are used otherwise. As the third contribution, the thesis makes a case for having an exit functionality in an HTM. The objective of the proposed functionality, `TM_EXIT`, is to terminate a transaction without restarting or committing.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487`), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see `http://www.manchester.ac.uk/library/aboutus/regulations`) and in The University's policy on presentation of Theses

Acknowledgements

I would like to thank Prof. Ian Watson, my supervisor, for accepting me as a PhD student to the APT group. He not only accepted me as a student, but also provided me with a scholarship that covered my tuition fees and living expenses. The whole PhD was funded by an Overseas Research Studentship (ORS) and a School of Computer Science studentship from the University of Manchester. I am very grateful to them, none of this would have been possible if I was not able to secure that scholarship. Prof. Watson did more than supervising, he helped me a lot in debugging and improving my writing. He gave me enough freedom to explore new ideas, listened well and always proposed a solution/suggestion when I was stuck in the dark. Dr. Mikel Luján, my co-supervisor, helped me in numerous ways throughout my PhD.

When I started my PhD, four years ago, I knew nothing about research. Adapting to the research environment was painful. The APT group (Jamaicans) was very helpful. Dr. Jeremy Singer (now a lecturer at Glasgow) and Dr. Ian Rogers (now at Google), as the research associates of the group at that time, helped me a lot in getting familiar with the lab and the research environment. Dr. Preethi Sam (now at ARM), Dr. Christos Kotselidis (now at Intel Labs), Dr. Behram Khan (now a research associate at Manchester) and Dr. Mohammad Ansari (now a lecturer at Umm Al-Qura University), as the senior PhD students of the group at that time, gave good insights about how to get on with the PhD. I would also like to extend my thankfulness to Dr. Daniel Goodman, for volunteering to present my paper at the Transact 2012 workshop, when I was unable to participate due to visa issues.

I would also like to extend my heartiest gratitude to the Disability Support Office (DSO) of the university and to the Environs officer of the Computer Science school, Mr. Eamon Griffin, for supporting me in every possible way they can for my health problems.

Friends are part of our lives. During my stay at Manchester, I have made good friends with whom I have shared moments of joy, sadness, frustration, anger (the list

can go on and on). Paraskevas (Paris) Yiapanis always had something to talk about. He is one of the best social beings I have ever known. Demian Rosas, my research partner, is a good listener. I have wasted lots of his useful hours with my “fantastic” ideas. The three of us used to have one to two hours of “lunch hours” discussing non-CS issues like world politics, economy (again the list can go on). I will always remember those nice times.

I would also like to thank my former supervisors Dr. Manjula Sandirigama, Dr. Swarnalatha Radhakrishnan and Dr. Roshan G. Ragel from the University of Peradeniya, for supporting and encouraging me to embark on a research career. Dr. Ragel was like a friend to me. He checked my applications, helped to improve research proposals, even let me use his credit card to pay the application fee.

My father, Mr. Dhanapal Herath, always valued education. He tried to give the best possible education to his kids. If I needed something for my studies (books, exam papers or anything), he made sure that I would have it by the time I needed it. He told me the importance of higher education and convinced me to read for a PhD. The unconditional love of my mother, Mrs. Wimala Dhanapala, was a magnificent source of inspiration to me. I thank my parents and my sister for being so loving and present even from a distance.

It is the Sri Lankan culture that everybody belongs to big family. All my aunts, uncles, brothers and sisters (I don’t know how many, its that big!) sent me well wishes whenever they can. I never felt alone. My lovely in-laws were in touch with me all the time, energizing me with Sri Lankan sweets.

Finally, and certainly not least, a great debt of gratitude is owed to Sashila, my wife, for her understanding and love. You were with me in every single step of this journey. You tried your best to make it comfortable for me. I love you.

Chapter 1

Introduction

Technology enables the doubling of transistors every 18-24 months (known as Moore's law [76]). This has allowed designers to increase complexity, by including more functional units in a processor, in search for higher performance. The objective of increasing the functional units is to allow a computation to proceed without waiting for resources. Having more resources facilitates instructions to be executed in parallel, which is therefore known as Instruction Level Parallelism (ILP). However there exists a limit to the number of instructions that can be executed in parallel [107]. In addition to this, the scaling limitations of single-core computing (power consumption and heat dissipation) [81] urges the hardware designers to investigate other directions.

In an attempt to address these issues, hardware manufacturers consider developing a processor with more than a single core on the same die, which is called a chip-multiprocessor or a multi-core processor. In this multi-core design, the speed of each individual core is less than the speed of a modern single core processor. As this multi-core processor comprises many of those average speed cores, an application which is executed on them is expected to complete in less amount of time than in a single core processor, or at least in theory that is what should happen.

Just because a processor has higher number of cores, does not guarantee that any application runs on it will deliver higher performance. A best case scenario would be, executing an application which comprises several independent work units which can execute on their own. In such a situation, each work unit can be scheduled in one core and the output can be produced in considerably less amount of time than running each work unit sequentially in a single core processor. However, this is not the case for most of the applications. In order to gain the advantage of all the cores in this multi-core processor, first an application needs to be divided into several work units. As the

execution of this application progresses, some work units may need to communicate with each other. Sometimes they may try to access the same set of variables at the same time. Therefore a considerable effort is required when transforming a single core application to take the advantage of a multi-core processor. Changing the direction of processor design from single-core to multi-core has introduced several issues and the most significant and most relevant of them to the scope of this thesis is summarised hereafter.

The data transfer rate between the processor and the memory is much slower compared to the speed of a modern processor. This creates a processor-memory gap, which is called the Von Neumann bottleneck. A small but faster storage called cache is attached to the processor to address this, by storing frequently used data in cache and servicing memory requests with them. Even though this was a promising way of achieving its objective, in a multi-core environment this can lead to a situation where more than one processor has different values in their caches for the same memory location. This is possible because now the memory is being accessed/modified by more than one processor and the cache of one processor may not be aware of the changes made by another processor. Therefore a mechanism is required to communicate the state of a cache line in one core, to the rest of the cores. As the issue is related to maintaining a coherent view of caches, the associated communication is called cache coherence protocol. Quite a number of protocols each having different performance characteristics have been proposed. All these protocols require a data structure to be maintained in hardware (a modified cache) to store the state (modified, invalid or identical to memory copy) of cached entries and a messaging mechanism to communicate the state of the cache lines concerned.

In a single core processor, instructions are issued by a single entity. Therefore the order the effects of these instructions appear, is the same as the order they have been issued. Simply, a read operation to a memory location should return the last value written to that location. The definition of “last” is trivial in a single core processor as there exists only a single program order. Also having a single order allows certain compiler optimizations such as reordering of instructions and so on. However in an environment where multiple entities are issuing instructions, firstly, the ordering is not trivial. Secondly, some of the compiler optimizations may not match the behaviour expected by the application developer. Therefore a mechanism is required to ensure that the memory operations issued by all the processors correspond to some order and this order should match the behaviour expected by a programmer. Memory consistency

models are introduced to address this issue.

In the multi-core computing era, software applications need to be modified to utilise the extra cores available in a processor. The workload needs to be distributed to available cores when executing these applications. Therefore it is inevitable that parallel programming becoming the mainstream programming practise. In an application that is written to execute in parallel, the parallelism can exist either at task level or method level or thread level. (The discussion is made at the context of thread level parallelism, but the issues remain the same for other levels as well). When a program is executing in parallel, different threads may access/modify the same memory location. Programmers need to identify those situations and special mechanisms (locks, barriers) need to be introduced to the program to produce the correct result. This is the synchronisation issue in parallel programming. In order to ensure mutual exclusiveness is maintained among those concurrent accesses made to these memory locations, conventionally, locks are used. To execute a code region protected by locks, first the lock is acquired. It is released when the execution is completed. In order to acquire a lock, at the hardware level it is required to first check whether it is available. The acquire operation happens afterwards. Essentially this involves a read and a write operation. If the process of acquiring a lock is done using conventional read and write operations, in a multi-core environment it is possible for two processors to acquire the same lock at the same time. Therefore atomic instructions like `Test-And-Set` which allows to atomically read and modify a memory location are used to implement locks. The instruction pair `Load-Linked/Store-Conditional` (LL/SC) is similar to other atomic instructions, but comprises two instructions. This allows it to have intermediate operations between the load and the store operations. This feature makes LL/SC a good candidate for constructing other atomic instructions. For example a new atomic increment instruction can be constructed using this LL/SC instruction.

The simplest form of lock based programming is to have a single program-wide lock and each thread acquires it before entering the critical region and releases it, once it has finished executing the critical region. This approach is quite simple and easy to explain. However this kind of approach is not advisable as there can be groups of critical regions where these groups have no relation to each other. Therefore despite its simplicity, this kind of coarse grained locking approach can harm the performance of a parallel application. The other alternative is to associate a smaller lock with each significant computation inside the critical region. In this manner if all the associated locks are available, a thread can complete the computation enclosed in the critical

region. Also in this case, non-availability of a lock means that the computation can not be performed in parallel. Even though this kind of finer grained locking seems promising in delivering better performance, writing applications with this approach is a difficult task. A programmer has to ensure that the program does not end up in a situation where two or more threads are waiting for each other to release a lock. This kind of situation is known as a *dead lock* and it can occur in any lock based program, if the locks are not managed properly.

Observing how optimistic concurrency is exploited in databases [37], Herlihy and Moss proposed a lock free synchronisation mechanism called Transactional Memory (TM) [50]. In the TM approach, instead of waiting for a lock to be available, all the threads enter the critical region as if the lock is available. In order to ensure the consistency and the correctness is not compromised, all the operations performed in this critical region are made speculatively and kept in isolation from other threads. Under TM, a critical region becomes a transaction and each transaction appear to rest of the system, as if it happened atomically. This is achieved by operating speculatively within the critical region and keeping those operations in isolation. This requires a mechanism to keep two versions (speculative and original) of memory locations that are accessed within a transaction. If more than one thread has accessed the same memory location and one of them has modified it, a conflict has occurred. In order to check conflicts, each thread is required to maintain a set of locations it has read and written in the current transaction. Based on how this versioning and conflict detection is performed several flavours of TM systems exist. Also all these TM related operations can be performed in hardware, software or using a combination of both (hybrid).

Having described the issues of multi-core computing related to this thesis, the discussion is now aimed to determine the root cause of these problems. Starting with a very general approach, a computer consists of large number of transistors, wires and so on. In order to make use of this equipment an application program needs to be developed. In order to facilitate the job of an application program developer, an abstraction of the underlying system is defined. Programmers write their programs according to the abstraction provided and this comes in the form of a programming model. In single-core computing this could simply be the Von Neumann model, since it comprises only one processor and one memory. Therefore programmers can only specify “what” operations need to be done with the memory and not how to do it. Due to its simplicity, programmers would still favour this kind of abstraction, even in the multi-core era. A problem occurs when ensuring a Von-Neumann like abstraction in

a multi-core processor, because the memory can now be modified by more than one processor. In order to provide this simple abstraction, at the architectural level, it is required that each processor has the same view of the shared memory. In this thesis, the term "global view requirement" is used to represent this. The simplest form of providing this is to ask each processor to issue instructions in an order, one after the other and the effects of each instruction becomes visible to the others before the next instruction is issued. In this regard, a cache coherence protocol can be used to propagate the effects of the execution of an instruction and a memory consistency model defines when this propagation should be done.

This kind of approach will certainly be able to provide a Von-Neumann like abstraction for a multi-core processor. However, it will restrict the optimizations like instruction reordering and bypassing and also incurs quite a lot of communication. Analysing this issue from a high level programming language perspective, it is clear that not all the threads are interested in the modifications made to all the memory locations by other threads. In terms of a coherence protocol this translates to, a propagation not being required for all the cache lines that get modified. The same observation, in terms of a memory consistency model can be interpreted as, based on the access pattern certain memory locations can be reordered or bypassed even without the programmer intervention. Even though the latter is provided in certain relaxed memory consistency models, a programmer is required to categorise memory locations to relax the ordering. Based on these observations one part of this thesis presents an architecture that relaxes coherence and consistency of memory locations based on their access patterns.

Going back to the discussion of abstraction, when providing synchronisation using locks, programmers have to define not only "what" data needs be synchronised but also "how" to provide it. It becomes the responsibility of a programmer to acquire and release locks in a way that the program does not end up in a *dead lock* state. This clearly breaks the much believed concept of abstraction in computing. The TM approach to maintaining synchronisation, is able to provide the same abstraction as it only requires the definition of the critical section and the underlying TM mechanism takes care of how the mutual exclusiveness is maintained among accesses to that critical section. Since TM relies on speculation, aborts can happen due to mispeculation. Also aborts can happen due the lack of clarity in the mechanism used to detect conflicts (false aborts). Another part of this thesis proposes a novel technique to reduce these false aborts.

It is the job of a programmer to mark the critical sections in a TM application and

the underlying TM system guarantees that all these critical sections will be executed speculatively and committed at the end. Committing requires communicating the speculative modifications to others. If there is a condition inside a critical section, in some cases it possible that all the prior computation becomes non-useful if this condition is not met. As the underlying TM mechanism is not aware of any such condition, a commit phase will always take place as it is the only way to complete the transaction. Another part of this thesis recognizes that, in those circumstances, there is a need to exit from a transaction without committing it, hence such a functionality is proposed.

1.1 Contributions

At a high level, this thesis makes three major contributions. The first contribution is the Data Centric Transactional Memory (DaCTM) in which the region based coherency and consistency is introduced. The basic idea of DaCTM is to group together memory locations of similar access patterns and to allocate them to different memory regions. The required level of coherence and consistency for each memory region is defined according to the access pattern of the locations it holds. The key idea is that, a location itself defines the required level of coherence and consistency for that location, which is the basis of the "data centric" concept. DaCTM is inspired by the "Transactions Everywhere" [60] approach and Transactional Memory Coherence and Consistency (TCC) [39] which suggest the development of an application entirely from transactions. The attractive component from those proposals is that transactions can be considered as the basic unit of parallel work, maintaining coherence and consistency, therefore provides a simple abstraction. However this simplicity comes at the cost of a higher bandwidth requirement which is not desirable in a multi-core environment. This is addressed in DaCTM by coupling this sort of "transactions-only" approach with the data centric concept. Using this approach, DaCTM is able to provide the same simplicity as "Transactions Everywhere" [60] and TCC [39] without saturating the communication network.

The second contribution of the thesis is SnCTM, a novel way of reducing false transaction aborts in a hardware TM system. The key idea is to use either cache lines or signatures to detect conflicts in a TM system, depending on the situation. SnCTM is motivated by two facts: (1)signatures produce false aborts; (2)signatures are only required when a transaction cannot fit in the cache. Therefore the proposal of SnCTM is to use cache lines to detect conflicts when a transaction is able to fit in the cache,

signatures are used otherwise. By adaptively changing between these two sources, SnCTM is able to reduce false transaction aborts in a hardware TM system. Also SnCTM is able to reduce the size of a signature without comprising the performance.

As the third contribution, the thesis makes a case for having an exit functionality in a hardware TM system. The objective of the proposed functionality (`TM_EXIT`) is to exit from a transaction without committing it. Once exited, the program control is transferred to the line immediately following the transaction. The proposal is motivated by the fact that when there is a condition inside a transaction, the whole computation can be non-useful when this condition is not met. However, regardless of the fact that the condition is met or not, a commit phase takes place. If there is a mechanism to notify that the commit is not useful, it could skip that step and go to the line immediately following the atomic block. The exit functionality proposed in the thesis fits well for this purpose. By avoiding these unnecessary commits, the network utilization can be reduced. In addition to making the case for this functionality, ways to improve expressiveness using this approach are also discussed in the thesis.

1.2 Thesis Structure

The contributions made in this thesis are centred around hardware transactional memory and each of them is independent of the others (as shown in Figure 1.1). Therefore the thesis is organized in to three parts.

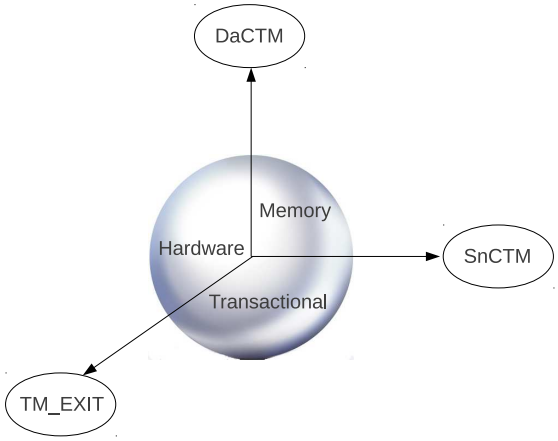

Figure 1.1: Contributions of the Thesis

Part I presents the Data Centric Transactional Memory (DaCTM) and comprises

Chapters 3, 4, 5 and 6. The concept of DaCTM is presented in Chapter 3. The architectural extensions required to support the DaCTM concept are described in Chapter 4. The performance evaluation of DaCTM is presented in Chapter 5. The other related work of DaCTM, except TM, is presented in Chapter 6. The detailed contributions related to DaCTM are described in the first chapter of Part I.

The SnCTM approach to reduce false transaction aborts is presented in Part II which comprises Chapters 7 and 8. The motivation and the concept of using adaptive sources to reduce false aborts is presented in Chapter 7. The same chapter also summarises related work on hardware signatures. The architectural design of SnCTM and its evaluation is presented in Chapter 8. A list of contributions made with the SnCTM approach is presented in the first chapter of Part II. The text in Part II is mostly based on [45].

Part III which comprises Chapters 9, 10 and 11, makes a case for having an exit functionality in an HTM. The motivation for the proposed functionality, `TM_EXIT`, is presented in Chapter 9. The same chapter formally defines the `TM_EXIT` function and describes how to use it in TM programming. Even though increasing performance is not the prime objective of `TM_EXIT`, the effect of it towards the execution time is presented in Chapter 10. Related hardware and software TM approaches that could either deliver or one would think is able to deliver the same functionality as `TM_EXIT` are discussed in Chapter 11. All the contributions made by introducing `TM_EXIT` are described in the first chapter of Part III. The text in Part III is mostly based on [44].

In addition to these three parts, Chapter 2 describes the background of Transactional Memory (TM). It focuses on conceptual aspects such as concurrency control, versioning and conflict detection; theoretical aspects such as syntax and semantics of TM; and implementation aspects such as hardware TM, software TM and Hybrid TM. As the thesis is centred around hardware TM, a significant portion of the chapter is dedicated to describe some of the key HTM proposals. The conclusions of the contributions made in the thesis is presented in Chapter 12. The same chapter also discusses the possible future research directions of each high level contribution.

Figure 1.2 shows the suggested reading order for the thesis. As each part is independent, the reader can only read the relevant chapters of an interested part and jump directly to the conclusions chapter. Also if the reader is experienced in the area of Transactional Memory, Chapter 2 can be skipped as each part has a separate chapter-/section describing most of the related work to the contributions made in that part.

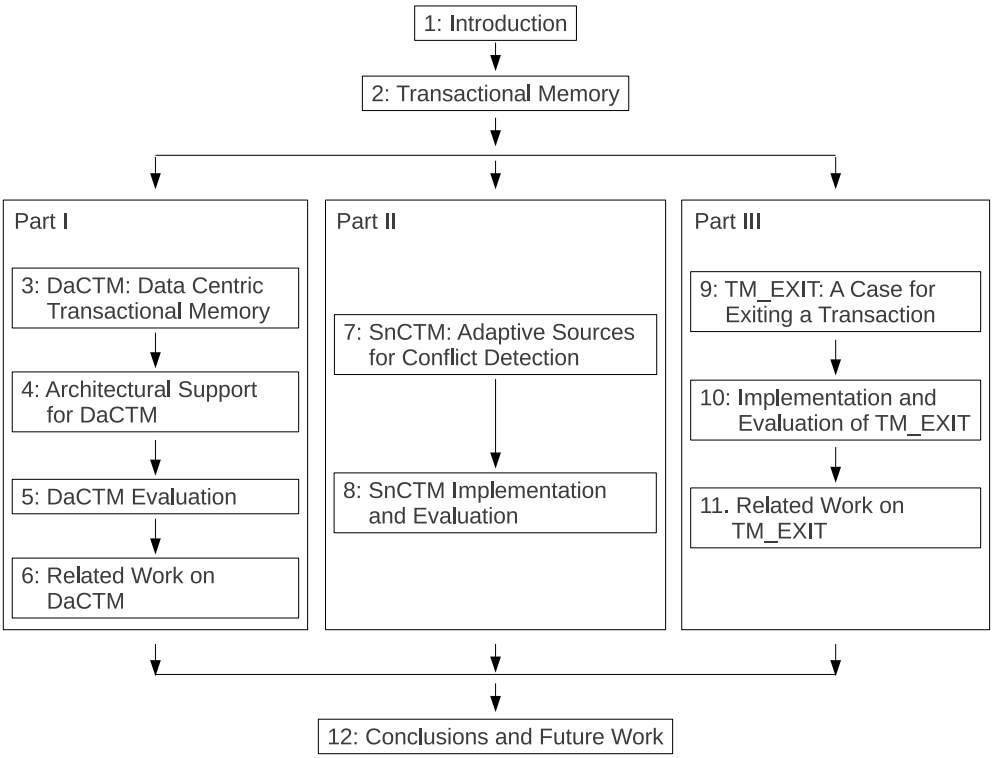

Figure 1.2: Suggested reading structure of the Thesis

1.3 Publications

The work presented in thesis has resulted in following publications.

- Isuru Herath, Demian Rosas-Ham, Mikel Luján, and Ian Watson. SnCTM: Reducing False Transaction Aborts by Adaptively Changing the Source of Conict Detection. In Proceedings of the 9th Conference on Computing Frontiers, CF '12, pages 65-74, New York, NY, USA, 2012. ACM.

- Isuru Herath, Demian Rosas-Ham, Daniel Goodman, Mikel Luján, and Ian Watson. A case for Exiting a Transaction in the Context of Hardware Transactional Memory. In TRANSACT '12: 7th ACM SIGPLAN Workshop on Transactional Computing, February 2012.

Chapter 2

Transactional Memory

The contributions made in this thesis are centred around Hardware Transactional Memory (HTM). In order to facilitate the reader to understand the contributions, to compare and contrast them with the existing proposals, this chapter provides a comprehensive summary of hardware transactional memory. It also gives an overview of software and hybrid transactional memory systems. The chapter starts the discussion by elaborating on theoretical and semantical aspects of Transactional Memory (TM) in Sections 2.2 and 2.3 respectively. Design considerations that could affect the performance of a TM system are discussed in Section 2.4. Advantages, issues and models for programming with TM are discussed in Section 2.5. A comprehensive summary of HTM based on the most influential existing proposals is presented in Section 2.6. Brief overviews of software and hybrid TM systems are provided in Sections 2.7 and 2.8. Finally, Section 2.9 summarises the chapter.

2.1 Introduction

When writing parallel programs mutually exclusive accesses are required in certain cases. Conventionally, a programmer may use locks to ensure that the mutually exclusive execution is guaranteed. In order to acquire a lock, at the hardware level it is required to first check whether it is available. The acquire operation happens afterwards. Essentially this involves a read and a write operation. If the process of acquiring a lock is done using conventional `read` and `write` operations, in a multi-core environment it is possible for two processors to acquire the same lock at the same time. Therefore atomic instructions like `Test-And-Set` which allow to atomically read and modify a memory location are being used to implement locks. The instruction

`Load-Linked/Store-Conditional` (LL/SC) is similar to other atomic instructions, but it also gives the possibility to construct other atomic instructions. For example a new atomic increment instruction can be constructed using this LL/SC instruction. However the atomicity is still maintained only for a single memory location.

Herlihy and Moss [50] propose to generalise this LL/SC instruction in order to provide atomicity to more than a single memory location. Their approach is called Transactional Memory (TM) which is based on the concept of database transactions [37]. They also inherited the concept of *atomic* blocks proposed by Lomet [69] as a notion of structuring a program. The initial TM proposal of Herlihy and Moss introduced a new multi-core architecture that is capable of providing lock-free synchronization. A programmer is expected to mark any number of instructions that need to be executed atomically. The atomic execution on multiple memory locations is made possible with speculation. When the end of the atomic region is reached, these modifications are communicated to other processors. The idea of executing code blocks and checking conflicts at commit points has also been expressed by Knight [57]. In the architecture proposed by Knight, all the modified cache entries are kept in a second cache called a *confirm* cache. Once a block has completed its execution, entries in the *confirm* cache are written back to the main memory. During this process, any other processor which has accessed these locations gets aborted.

Transactional Memory and database transactions have certain similarities. For example, some of the properties that a database system needs to maintain, are also required to be maintained in a TM system. These are known as ACI (Atomicity, Consistency and Isolation) properties. A transaction in TM context encompasses one or more memory operations. *Atomicity* requires a transaction to either complete all its memory operations or to leave the system as if none of those took place. If a transaction is successful, it commits all its speculative operations thereby making them visible to the rest of the system. If a transaction is not successful, it aborts thereby abandoning all its speculative modifications and leaving the system unmodified. *Consistency* requires a system to be transferred from one consistent state to another consistent state. In other words it ensures no transaction leaves the system in a half-finished state. Finally the *Isolation* property requires all the modifications made within an atomic region to be kept in isolation until the commit point. Modifications made by one processor are not visible to others until they are committed.

2.2 TM Design Considerations

When executing transactions, *conflicts* occur when more than one processor accesses the same memory location and one of them is a write. These conflicts then need to be *detected* and *resolved*. The detection and resolution of conflicts can be performed at the time the conflict occurs or they can be deferred until later in the execution. The concurrency control of a TM system defines how these events are handled.

2.2.1 Concurrency Control

Currently there are two basic approaches to concurrency control. In the first approach, known as *pessimistic concurrency*, all three events happens at the same time. That is when a transaction is about to access a memory location, it tries to get the exclusive ownership of the location. If another transaction has already accessed this location a conflict is detected and, depending on the conflict resolution policy, one of the transactions is aborted immediately.

In the second approach, known as *optimistic concurrency*, these events can happen at different times. For example multiple transactions can modify the same memory location. Even though this results in a conflict, it is not detected until one of the transactions decides to commit. When the committing transaction publishes the locations it has accessed, the TM system detects conflicts and resolves them.

It is hard to advocate which form of concurrency is better because it depends on the nature of the application. If the application has high contention, it tends to produce more aborts. For such situations *pessimistic concurrency* becomes useful because it help to reduce the wasted work. However it requires exclusive ownership before accessing a memory location. This could lead to a live-lock situation, which affects the forward progress of the execution. On the other hand *optimistic concurrency* does not require such ownership, allowing any number of speculatively modified entries to exist for the same memory location. Therefore it does not introduce any live-lock and it guarantees at least a single thread will progress.

In addition to these two basic mechanisms, there is a third alternative of eager detection but delayed resolution. In this scheme, conflicts are detected as they occur. However, which transaction to abort is decided later, most probably when an affected transaction is committing. The advantage of such a mechanism is that, occurring of live locks in a *pessimistic concurrency* control system can be avoided as in *optimistic concurrency* , by delaying the conflict resolution. At the same time this mixed mode

has the ability to detect conflicts early as in *pessimistic concurrency*.

2.2.2 Version Management

In order to ensure the atomicity and isolation properties of TM, it is required to maintain the original and speculatively modified versions of memory locations that are accessed within a transaction. The *version management* of a TM takes care of this. In the first approach, *lazy versioning*, all the speculative operations are performed on a local copy thereby keeping the original memory location unmodified throughout the execution of a transaction. Once the execution of the transaction is finished, all the original memory locations are updated with the speculatively modified values.

In the second approach, *eager versioning*, the original memory locations are modified during the execution of a transaction. Since the original memory location is modified, the old value needs to be recorded in a separate log. This is important because if a transaction is not successful it should leave the system as if nothing has happened. Since the original memory locations are modified in this approach, they are restored using the values from this log in such situations.

Again it is hard to decide which is better, because each approach has its advantages and disadvantages. The overhead of an abort operation is negligible in *lazy versioning* because it only requires to clear the local copies of memory locations as the original memory locations remain unmodified during the execution of a transaction. However a commit operation requires to explicitly update original memory locations. Therefore *lazy versioning* is better suited to applications in which aborts are frequent. On the other hand the overhead of a commit operation is negligible in *eager versioning* systems because when a transaction is completed, all its speculative modifications are already published. Aborts are costly for these systems as they require to first access the log file and to revert all the modifications made to the memory locations during the atomic execution. For applications which have fewer aborts, *eager versioning* is better suited.

2.2.3 Conflict Detection

When more than one transaction accesses the same memory location and one of them is a write, a conflict has occurred. A TM system employs a *conflict detection* mechanism to identify conflicts. This can be categorised based on the time of the detection and granularity of the detection. If conflicts are detected at the same time they occur, then

it is called an *eager conflict detection*. In such a system when a transaction is going to access a memory location, it is checked to see if any other transaction has already accessed this data in their current transaction. If that is the case, one of the transactions has to abort or wait until the other one completes. In *lazy conflict detection*, conflicts are checked when a transaction commits. There, when a transaction commits it notifies the TM system about the memory locations it has modified and the TM system checks whether any other transaction has accessed these locations.

When considering the granularity of conflict detection, hardware TM systems can check the conflicts either at word level or at the cache line level. When conflicts are checked at cache line level false conflicts can occur due to cache line sharing. This can be avoided if conflicts are detected at word level granularity, then a cache line will have certain valid words and invalid words. Therefore extra care is required to avoid using invalid words. In software TM, conflicts can be detected at object level granularity or at word level granularity.

2.3 TM Semantics

This section describes semantics of TM systems. It describes what properties are required to maintain in a transaction and how the transactional and non-transactional accesses are handled. The section also describes several platforms for defining characteristics of an atomic block. Different classes of transactional nesting are covered at the end of the section.

2.3.1 Serializability

Serializability requires that the effects of a transaction becomes visible to the rest of the system as though they had executed in a serial order. That does not mean the ordering should be the same as they execute. For example if a system has two transactions T1 and T2, the effects of them can be visible either as T1$\rightarrow$T2 or T2$\rightarrow$T1. Relaxing the ordering could be advantageous in certain cases. For example if the operations associated in the next transaction have no relation to the current transaction, the order they becomes visible can be relaxed. However if the second transaction depends on the data produced by the first one, relaxing the order can lead to inconsistencies.

2.3.2 Strict Serializability

Strict Serializability is a stronger requirement than serializability. This requires that the order, the effects of transactions become visible to others should be the same as the order they execute. For example if the transaction T1 executes before T2, then the effects of T1 should be visible to the others before T2.

2.3.3 Linearizability

Linearizability requires that a transaction appear to have completed all its operations at a single point in the program order. This emphasise that all the speculative operations should atomically become visible to others. Reasoning about TM execution is made easier with linearizability because all the read write operations inside an atomic block can be represented by a single operation in the program order.

2.3.4 Weak Isolation

In a program there can be transactional and non-transactional accesses, even to the same memory location. Weak Isolation (WI) guarantees the TM semantics only among transactional accesses. Even though this reduces the overhead on a TM system, in certain cases this can be problematic. For example consider a situation where a transaction reads the same memory location several times and a non-transactional write is made to the same memory location between those reads. If WI is in place, this produces an inconsistent view of the memory as the latter reads observe the updated value whilst early reads see the old value within the same transaction.

2.3.5 Strong Isolation

The above problem can be addressed with Strong Isolation (SI) which guarantees TM semantics among transactional and non-transactional accesses. When SI is maintained, transactions are isolated from other transactions and from any other non-transactional operations. Therefore in the previous example, a transaction will not see the update by the non-transactional write. It might either signal a conflict or the non-transactional write will be delayed until the transaction commits.

2.3.6 Single Lock Atomicity

Single Lock Atomicity (SLA) is a model that can be used to define semantics of a transaction with respect to other transactional and non-transactional operations. Under SLA, the behaviour of a transaction is defined as if there is a program wide global lock. For example under SLA, the execution of T1 and T2 transactions will be similar to as if one of T1 or T2 acquiring the global lock and finishing the transaction and thereafter the other one doing the same. SLA model is quite simple and also programmers can easily become familiar with it as it extends the lock based programming model to TM. Despite its simplicity, SLA cannot be used to define the behaviour of all the situations that occur in TM programming. One such problem occurs when defining the behaviour of nested transactions. This is because, according to the definition of SLA, if there is only one program wide lock, the inner transaction has to wait indefinitely as the outer transaction has already acquired the global lock. Another situation arises when a transaction enters an infinite loop thereby preventing the progress of all others transactions as they cannot acquire the global lock.

2.3.7 Disjoint Lock Atomicity

Some of these issues have been addressed in Disjoint Lock Atomicity (DLA), which is a weaker model than SLA. Under DLA, there is no global lock, instead a transaction is required to acquire a set of locks corresponding to the data that it intends to access. Therefore if two transactions access disjoint data, they do not need to wait for each other. For example, in order to execute transaction T1 under DLA, it only requires to acquire locks related to the memory locations it intends to access. If the other transaction T2 does not access the same set of memory locations, they can execute in parallel.

2.3.8 Transactional Sequential Consistency

Transactional Sequential Consistency (TSC) proposes to define the semantics of a transaction using TM itself. TSC is derived from extending Sequential Consistency (SC) to the TM domain. Under TSC, transactions appear to have happened atomically without any interleaving with other operations in the system. TSC stands as a better model than any lock based models, because it allows a programmer to precisely reason about the outcome of an application which has transactional and non-transactional accesses.

2.3.9 Nested Transactions

A transaction becomes nested if it is inside another transaction. In such situations it is not straightforward to define the behaviour of a nested transaction. Typical questions which arise with nested transactions are whether the commit of an inner transaction becomes visible to other transactions or whether the abort of an inner transaction aborts the outer transaction and so on. Different approaches to handle nested transactions are summarised below.

2.3.9.1 Flattened Nesting

This is the simplest form of nesting. The inner transaction becomes a part of the outer transaction. Therefore committing the inner transaction does not make its changes visible to the others until the outer transaction commits. Similarly if the inner transaction gets aborted, the outermost transaction gets aborted as well.

2.3.9.2 Closed Nesting

In closed nesting, when an inner transaction gets aborted it does not affect the outer transaction. If the inner transaction commits, its changes becomes visible to the outer transaction. However these changes do not become visible to the rest of the system until the outermost transaction commits. If the inner transaction commits the behaviour of both flattened and closed nesting are the same. However the operations involved in closed nesting are more costly than the flattened nesting. Therefore if commits are frequent flattened nesting will outperform the closed nesting. Conversely if aborts are frequent closed nesting will prevent aborting the entire transaction, thereby performing better than flattened nesting.

2.3.9.3 Open Nesting

In open nesting, when an inner transaction commits, its changes become visible to the rest of the system. Therefore even if the outer transaction gets aborted, the modifications made by the inner transaction are kept committed. Open nesting could be used to improve concurrency by treating the inner transaction as a separate transaction. However extra effort is required when using open nesting as the isolation property of the outer transaction can be dropped by the inner transaction, thereby leading to inconsistencies.

2.4 TM Performance Considerations

Programming with locks can harm the performance of a parallel application as it is a blocking operation. One of the motivations for proposing TM is because it is a non-blocking operation. As there can be different ways of providing non-blocking synchronization, this section first briefly summarises them. The section also covers another performance consideration which is the contention management of a TM system. Finally the section also describes *early-release*, which is an optimization proposed to reduce conflicts, as they can affect the performance.

2.4.1 Non-blocking Synchronization

The first consideration of TM performance is the extent to which it supports the liveliness of an application. Liveliness is reflected by the amount of progress which is guaranteed in an application when using TM. In other words, whether the synchronization provided by TM is *non-blocking*? An algorithm can be considered as *non-blocking*, if a pre-empted operation does not block the other operations making progress. Several variations of *non-blocking* synchronization exist. The strongest one is *wait-free* [46] in which each transaction is guaranteed to complete in a finite number of steps regardless of the actions of other transactions. A slightly weaker criterion is *lock-free* [47] in which some transactions are required to finish in a finite number of steps. Herlihy *et al.* proposed an even weaker criterion for *non-blocking* synchronization which is *obstruction-free* [48]. Synchronization techniques belonging to this type only guarantee forward progress when there is no contention. This is able to provide a simplified implementation of TM algorithms whilst delivering the benefits of *wait-free* and *lock-free*.

2.4.2 Contention Management

Another performance consideration is what sort of contention management (CM) policies are supported in a TM system. CM is responsible for deciding the best course of action when a transaction aborts. The simplest policy is to abort a transaction whenever it encounters a conflict. This can result in aborting the same transaction multiple times, for example, if another transaction is operating inside a loop and modifying a shared variable. In some cases, introducing a delay with an exponential backoff can solve the problem. Another solution is to give priority to transactions based on the number of

operations performed. In such a scheme, when a conflict occurs the transaction with a lower priority aborts and the higher priority one continues. This can lead to starvation. It is hard to define which policy is the best, because it depends on the characteristics of the application. Several CM policies have been proposed and a good survey of them is presented by Scherer *et al.* [97].

2.4.3 Early Release

Another way to improve performance in a TM system is to reduce the number of conflicts. Several implementation techniques can be employed to achieve this. This can also be achieved by extending the TM system to allow a programmer to manually define certain memory locations as non conflicting regardless of being accessed inside transactions. Early-release provides this kind of functionality [49, 100]. The most common example to demonstrate the use of early release is when one transaction is performing a search and another one is modifying a part of the data structure being searched. This can lead to conflicts even if both do not focus on the same data item. With early release, a programmer can explicitly remove entries from the read set of the transaction. For example in the above scenario, the entries accessed in the search method can be removed thereby reducing unnecessary conflicts. However using *early-release* requires great care as it can compromise the correctness, if variables are removed from the read set erroneously.

2.5 Programming with TM

Even though the initial goal of TM was to provide atomicity for more than a single memory location, it has grown substantially as a programming model with particular emphasis on the exploitation of future multi-core architectures. This section describes the advantages of TM over lock based programming. It also discusses certain functionalities that either cannot be provided or are difficult to provide with TM as opposed to lock based programming. Finally it also describes a programming model which is completely based on TM.

2.5.1 Composability

One of the advantages provided by TM is that the programmer is no longer required to name the resource that the synchronization is based on. This is different to lock

based programming in which programmers have to state the lock on which the synchronization is based on. In TM, they can simply state that a method or a block of code needs to be synchronized. Programming is made easier with TM because it only require programmers to define what to synchronize, not how to do it. The indirect advantage of this is the composability. That is with TM any number of atomic blocks can be combined together in order to form another atomic block. This does not require any programming effort, whilst achieving a similar outcome with lock based programming requires great care.

2.5.2 Conditional Synchronization

One of the issues that a programmer faces with TM is how to provide conditional synchronization. For example consider a situation where one transaction has to wait if a data structure is empty and another transaction is pushing items in to this data structure. With lock based programming this can easily be achieved by asking the first thread to wait on a condition variable (eg: `pthread_cond_wait` [54]). A similar approach cannot be applied in TM programming as the operations performed within an atomic block are done in isolation. Harris *et al.* [43] proposed a `retry` statement to support this kind of scenario. The underlying functionality of `retry` is to transfer the control to the beginning of the same atomic block which has invoked it. Simply, the same transaction is retried using `retry`. For example consider a situation where a pop function has to wait until a list is not null. In this case, a functionality similar to conditional waiting can be delivered with `retry`. That is by issuing a `retry` statement when the list is null. Thereafter the same transaction is retired until the list is not null, hence similar to waiting for the list to be not null.

2.5.3 Memory Allocation

Another issue that needs addressing is memory allocation. The first problem with memory allocation is, it can be a serializing operation. Imagine a situation where a memory allocation is happening inside an atomic block. As memory allocations are served serially, all the concurrently running transactions also become serialized. This could ruin the whole concept of TM. However this can be addressed by having a parallel memory allocator similar to Hoard [6]. A second problem arises when defining the behaviour of speculative allocation. If TM semantics are strictly enforced, then the allocation should not be visible to other threads until the commit point of the requesting

transaction. That means, for example, if there were 5 free blocks and a speculative request is allocated 3 blocks, the rest of the transactions should still see that 5 blocks are available, not 2 (5-3). This means the memory allocator cannot keep a precise count on the available memory, because the speculatively allocated memory cannot be given to any other request because then it will overwrite the values written by the other atomic block. For example in the same scenario, if another transaction requests 3 memory blocks, the request cannot be serviced because there is no enough free blocks to serve that request. However looking at this denial from the requesting transaction's view, there is nothing stopping the allocator granting its request because, according to the requester's view, there are 5 free blocks available. Some of these issue has been addressed by Hudson *et al.* in their McRT-Malloc allocator [53].

2.5.4 Transactions Everywhere

Two major approaches can be used when integrating TM with parallel programming. The most straightforward way is to replace locks with atomic blocks. In certain situations this can lead to problems. If the code region protected by locks uses conditional synchronization, then the code needs to be modified to reflect the available functionality in TM to support such a feature. Another situation is, if the mutually exclusive code has certain irreversible actions or interactive input/output operations then the underlying TM specification should be studied to realise how the lock based code can be transformed to a TM version to perform similar to the original.

The other approach is to transform the entire program into a collection of transactions. The "Transactions Everywhere" approach proposed by Kuszmaul and Leiserson [60] belongs to this category. The objective of this approach is to free the user from managing complex synchronization protocols as the entire program can be seen as a collection of atomic blocks. As all the operations have to be performed within a transaction, there cannot be any data races within transactional and non-transactional accesses. The authors extend the Cilk [32] programming language with the `atomic` keyword to denote that a method or an operation followed by it needs to be transactional. In their approach transaction boundaries of an application are defined at places like return statements, spawning a thread and so on. If the atomicity is required to be maintained across these exit points, the `atomic` keyword is used explicitly. Transactional Memory Coherence and Consistency (TCC) [39] is an architecture proposal for using transactions as the basic unit of parallel work, maintaining coherence, concurrency and synchronization. The same authors also proposed a loop based and fork

based programming extensions to support the TCC architecture [38]. When a typical `for` loop is replaced with the proposed `t_for` construct, all the iterations of the loop are guaranteed to be executed as separate transactions. Different flavours of this construct allows to change certain parameters of the execution, such as controlling the number of iterations included in a transaction or maintaining an order among them and so on. They also proposed `t_fork` which allows the execution of a method call as a separate transaction while the callee method continues its execution from the line immediately following the method call.

2.6 Hardware Transactional Memory

Systems in which all the TM related operations (conflict detection, version management) are performed in hardware belong to the category of Hardware Transactional Memory (HTM). When compared to Software based TM systems, HTM systems can provide better performance. Also they do not require application rewriting. However they are not as flexible as software TM systems. Another disadvantage of HTMs over software TMs is that the limitations imposed by the physical hardware such as caches. For example, speculative data is stored in the Level 1 (L1) cache of a TM processor. This implicitly requires a transaction to be bounded by the size of the L1 cache. In addition, HTMs found it difficult to maintain speculative data among context switches and/or thread migration. This is because, in either case caches need to be evicted to the main memory before performing the associated operations. In an HTM system this cannot be performed directly, hence require further extensions. The literature on the area of HTM is huge, therefore summarising them all is beyond the scope of this thesis. Interested readers are directed to the Chapter 5 of the Transactional Memory book [42] by Harris *et al.*. This section describes some of the key proposals in HTM literature.

HTM systems can be categorised in several orthogonal axes. The first criterion is whether the HTM is implicit or explicit. The latter requires special read and write instructions to be used when accessing a location inside a transaction. Since new instructions are introduced, it is possible to treat only the necessary memory locations as transactional, even if they are performed within an atomic block. The advantage of this approach is that the read and write set of a transaction only contains memory locations which require ACI properties to be maintained. This results in a smaller read and write set, hence is useful in addressing the problem of resource limitation in HTM. The disadvantage of these explicit HTMs is that they require a TM application

to be rewritten to use new transactional read and write operations. This introduces a problem when using library functions or application code which has already been compiled. One solution is to have different versions for library routines, a TM version and a non-TM version. However this is not possible when the library is available only in binary format.

The TM system proposed by Herlihy and Moss [50] falls in to this category. In their proposal, programmers are equipped with three memory operations: `Load-transactional` (LT) -to read a memory location; `Load-transactional-exclusive` (LTX) -to read a memory location with the intention of modifying it; `Store-transactional` (ST) -to speculatively modify a memory location. All the memory locations accessed with LT become the read-set and locations accessed with LTX and ST become the write-set of a transaction. In addition they also introduced three operations to control the execution of a transaction. The `Commit` operation makes all the speculative updates visible to others by updating their original memory locations. All the speculative modifications are abandoned by calling the `Abort` operation. The last operation, `Validate`, is used to test the status of a transaction itself. If a transaction has aborted, it returns *False*, else it returns *True*. They do not provide an operation to indicate the start of a transaction, the first call to LT, LTX or ST is considered as the start of a transaction and is committed by explicitly calling `Commit`. Each processor is equipped with a transactional cache in addition to the regular cache and they are exclusive, meaning that an entry can only be residing in one cache. The transactional cache holds the speculative values which become visible to others on a commit or discarded on an abort. Transactions are aborted if an interrupt happens when a processor is executing a transaction. The same applies for transactional cache overflows.

The Oklahoma Update proposal by Stone *et al.* [104] is a synchronization mechanism that requires explicitly defined memory operations. Their proposal comes with three operation to manipulate data: `Read-and-Reserve` -reads a memory location and reserves it in a register called the *reservation* register; `Store-Contingent` -modifies the data after copying it to a reservation register, these modifications do not become visible to other processors; `Write-if-Reserved` -this takes multiple reservation registers as its argument and updates their original memory locations with the speculative data. Thereafter the speculative data becomes visible to other processors. When `Write-if-Reserved` is executed, it tries to acquire the ownership of the memory locations corresponding to the given *reservation* registers. Once all the ownerships have been obtained, all the locations are updated and ownerships are released thereafter.

The Advanced Synchronization Facility (ASF) [21] is a proposal from AMD to support HTM. It also falls into the category of explicitly defined TM systems. ASF proposes seven new instructions. The `SPECULATE` instruction starts a new transaction and the `COMMIT` instruction finishes it. The `LOCK MOV` instruction is used to transfer data between the processor registers and memory. This is similar to explicit read/write operations from Herlihy and Moss [50] and Oklahoma Update [104]. The `ABORT` instruction of ASF, explicitly aborts a transaction. In addition, system calls, exceptions, interrupts and eviction of transactionally modified entries from the cache cause a transaction to abort. The `RELEASE` instruction releases an entry from the read-set of a transaction. Thereafter it is not checked for conflicts. Finally, `WATCHR` and `WATCHW` detect stores and loads from other processors, to a set of given addresses. In ASF, when a transaction is aborted it jumps to the line immediately following the `SPECULATE` instruction. One option is to continue executing the atomic block. The other is to jump to an alternate location by manipulating the Zero flag. Using the latter approach, ASF can execute an abort handler and it allows the passing of an abort code which tells the handler the cause of the abort [23].

Most of the HTM systems belong to the category of implicitly defined ones. In this category a programmer is expected to only mark the start and the end of an atomic block, the underlying TM system treats all the operations within these two boundaries as transactional. The advantage of such an approach is that the TM application can use any external library without modifying it. The disadvantage of such an approach is all the memory locations accessed within an atomic block are considered transactional regardless of whether they require it.

Speculative Lock Elision (SLE) [87] was one of the to first to propose implicit hardware support for lock free execution using speculation. Event though the authors did not use the term “Transactional Memory”, it is basically similar to a TM system. No modifications are required to the application code in order to gain the advantage of SLE. In the proposed architecture, when a lock acquire operation is detected, the lock is not acquired, instead it is assumed that the lock is available and the execution is continued. All the modifications made within the lock region are buffered so that no other processor sees them. Existing cache coherence protocols are used to check whether any other processors are accessing the same location as this one. If that is the case, a conflict has occurred and the execution is restarted this time by explicitly acquiring the lock. If no conflicts have occurred, the buffered entries are committed atomically. The same authors proposed Transactional Lock Removal (TLR) [88] in

which locks in a lock based program are replaced with transactions and timestamps are used to detect conflicts.

Rundberg and Stenström [92] argue that having an order for entering and exiting the lock region in SLE [87] can harm the concurrency. Instead they propose to specify an order once all the threads have reach the lock release statement. Their proposal, Speculative Lock Reordering (SRL), works as follows. All the threads enter the lock region assuming the lock is available, similar to SLE [87]. Once all the threads have reached the lock release statement, a *thread dependency graph* is created. Using this graph a commit order is formed in order to minimize dependencies. Conflicts are detected at commit time in SLR, as opposed to eager conflict detection in SLE.

Hammond *et al.* propose Transactional Memory Coherence and Consistency (TCC) [39] in which transactions are considered as the basic unit of parallel work, communication, cache coherence and memory consistency. In TCC a program is decomposed into several transactions and the underlying architecture maintains coherence and consistency at transaction level, thereby providing a simplified programming model. However this simplicity comes at the cost of higher bandwidth requirements. Cache lines are extended to record read and write sets of a transaction. In order to maintain the isolation property of a transaction, speculatively modified entries are not allowed to be flushed during the execution of an atomic block. The authors propose to either use a victim buffer or to gain exclusive commit permission if the flushing is inevitable due to the capacity of the L1 cache.

Ananian *et al.*propose Unbounded Transactional Memory (UTM) [3], an idealised HTM design which supports the execution of unbounded transactions. The term "unbounded" encompasses transactions of arbitrary size and duration. Arbitrary size means that transactions can have a read or write set bigger than the L1 cache, in fact they can even be bigger than the physical memory but have to be less than the virtual memory. Arbitrary duration means that transactions can be longer than the time slice or the scheduling quanta. The first requirement is supported by having a structure called *xstate* in the memory and storing all the transactional information in that structure. If the size of this structure is not enough for a particular transaction, it is aborted and restarted after the operating system allocates a bigger area for the structure. By treating the *xstate* as a system-wide data structure and saving a pointer to this structure in the processor state, UTM is able to allow transactions to be longer than the scheduling quanta or even to migrate from one processor to another. Even though UTM is an

attractive proposal for executing unbounded transactions, it requires significant modifications to the processor hardware and to the memory subsystem. Therefore the same authors also proposed Large Transactional Memory (LTM) which is a simplified version of UTM. LTM can support transactions which have a read/write set bigger than the L1 cache, but they have to be smaller than the physical memory. LTM cannot support transactions which are longer than the scheduling quanta. In LTM when a transactionally modified cache entry is flushed while the processor is still executing the same transaction, the ejected entry is stored in an uncached area in the memory. The L1 cache is extended to have an additional 'O' bit to denote that a transactionally modified entry has been removed from the cache. When detecting conflicts both the L1 cache and this overflow area is examined.

Rajwar *et al.* argue [89] that most of the HTM proposals require programmers to be aware of system specific parameters such as the buffer size and the scheduling quanta, hence can be unattractive. They propose Virtual Transactional Memory (VTM) in which those system specific parameters are shielded from a programmer, similar to the virtual memory shielding the parameters of the physical memory. The idea is to decouple the TM from the underlying architecture by virtualizing it using several data structures. In VTM each transaction is associated with a *Transaction Status Word* (XSW) which acts as the sole authority of the associated transaction. The attractive component of the VTM is the *Transaction Address Data Table* (XADT), which keeps track of the overflowed memory locations. This gives VTM the ability to support transactions that exceed the size of the L1 cache, transactions that can be swapped from one processor and scheduled in the same or another processor and so on. They also proposed to include an *XADT Filter* (XF) to support fast execution of transactions that fit in the cache and do not encounter context switches. In VTM if a transaction is able to fit in the cache, it will be executed without using any of these data structures. When this is not the case, these data structures are used. However none of these structures are visible to the programmer, thereby making the programmer less worried about managing internal TM data structures.

Moore *et al.* argue [77] that most of the TM proposals use lazy version management, hence require buffering of all the speculative data which needs to be committed at the end, making it a slow process. Their proposal, Log-TM, relies on eager versioning and eager conflict detection. In Log-TM all the memory updates are performed in-place and the old values are stored in a per thread cacheable log. Therefore when a

transaction commits, it does not require to transfer any data as the speculatively modified entries are already updated. When a transaction gets aborted, this log file is walked and all the modified entries are replaced with old ones. Log-TM uses the existing cache coherence protocols to detect conflicts. They introduced a new *sticky-M* state which allows a processor to keep the ownership of a transactionally modified cache entry even after it has been evicted from the cache line. The advantage is that conflicts can be detected even for evicted cache lines without examining external data structures as in LTM [3] or VTM [89].

Page based Transactional Memory (PTM) is proposed by Chuang *et al.* [22] in order to support transactions that are not limited by the space and/or time. Even though their approach is similar to VTM [89], PTM maintains overflow information at page level granularity whereas VTM does it at the cache line granularity. In PTM if a transactionally modified cache entry is evicted before committing, a *shadow page* is allocated and original data is copied there. The modified data is stored either at the *home page* or at the *shadow page* depending upon the type of PTM (Copy or Select). As the pages (*shadow* and *home*) used in PTM are physical, as opposed to virtual data structures in VTM [89], no data is moved on commit. However on an abort, original data needs to be restored to the *home* page.

Ceze *et al.* propose [15] to use hardware signatures to represent the read and write sets of a transaction. A signature is a fixed set of bits, implemented using bloom filters [7], in which certain bits are set according to the address being considered. The important contribution from their approach, Bulk, is that read and write sets of a transaction do not need to be recorded in the L1 cache thereby allowing transactions to grow beyond the size of the L1 cache. In order to insert an address to a signature, certain bits are selected by hashing the address. In order to test whether an address is already in the signature, the same hashing is performed on the address and the bits are checked. The disadvantage of using signatures is that they can produce false positives, meaning when checked for membership they may assert positive even though it is not the case.

In order to simplify the modifications required to support unbounded transactions, Blundell *et al.* [9] propose *permissions-only cache*, a cache like structure that can record the speculatively modified cache lines that are evicted during the execution of a transaction. As their HTM uses eager versioning, updates are made in-place and the original values are recorded in a log as in Log-TM [77]. When a transactionally modified cache entry is evicted, the address is recorded in this *permissions-only cache*,

but not the data. By using an efficient encoding mechanism, the authors claim that a 4KB *permissions-only cache* can support up to 1MB of transactional data. They also propose OneTM, an HTM system which supports unbounded transactions, simply by restricting one overflow at a time. In OneTM-Serialized version, once a transaction is allowed to overflow all the other transactions have to stall whilst OneTM-Concurrent allows non-overflowing transactions to execute concurrently with the overflowing one. In order to facilitate conflict detection among overflowing and non-overflowing transactions, the authors use per-block meta data.

Following the proposal for encoding read and write set of a transaction to a fixed sized signature by Ceze *et al.* [15], LogTM Signature Edition (SE) was proposed by Yen *et al.* [110]. LogTM-SE decouples the version management and conflict detection of an HTM from hardware caches thereby making them virtualizable. Signatures are used to track read and write sets in LogTM-SE. The authors propose to use a *summary signature*, which is a union of all the signatures of threads that are currently inactive, to detect conflicts among active threads and inactive threads. As the software logs used by the LogTM-SE are accessible by the operating system, transactions can be migrated from one processor to another processor as simply as migrating a conventional process. Even though the logs are software accessible in LogTM [77], it cannot support virtualizable transactions as it relies on R/W bits in the cache lines to detect conflicts. This is because, in LogTM, conflict detection was not decoupled from caches..

Chafi *et al.* [17] propose to integrate a distributed directory structure to the TCC [39] proposal in order to provide a scalable architecture. Using this approach the authors were able to allow parallel commits which were not available in the original TCC design. This is made possible by introducing a *Sharing Vector* and a *Writing Vector* which keep track of the directories that the current transaction has read and written respectively. Even though Scalable TCC does not rely on conventional cache coherence protocols, it does not need to write-back data on a commit, as in original TCC.

Adapting the concept of *tokens* from *token coherence* [71], Bobba *et al.* propose TokenTM [10], an HTM system which relies on per block meta data to detect conflicts. In TokenTM each memory block is associated with T number of tokens. In order to perform a transactional read, a transaction is required to obtain a single token. All the T tokens of a memory block are required to perform a transactional write. Meta-data is stored in the memory by reusing some of the bits used for storing error correction and detection information. As meta-data is stored in memory, TokenTM can easily

virtualize transactions. The advantage of TokenTM over LogTM-SE in implementing virtualizable transactions is that, the former can provide precise conflict detection whereas the latter encounters a large number of false conflicts due to the use of signatures. As meta-data requires to record the an ID of the thread (TID) who has acquired the token, in order to make transactions virtualizable the TIDs need to be unique among all the processes.

HTMs systems which support eager conflict detection are capable of minimizing the wasted work and also require minor modifications to the existing cache coherence protocols. However, they require a good contention management mechanism to decide which transaction to abort as they occur eagerly. On the other hand, systems supporting lazy conflict detection do not have this problem, but they require a validation mechanism at the commit phase. The term conflict detection in this context actually refers to both the detection of the conflict and resolving it. All the proposed HTMs consider these two events together, hence performed at the same time. Tomić *et al.* [105] proposed to separate these two events, thereby taking advantages of both eager and lazy conflict detection. Their system, EazyHTM, detects conflicts eagerly, but the resolution of conflicts is deferred until the commit time. By using the conventional cache coherence protocol, each processor tracks conflicts eagerly and maintains two lists: *Races-list* and *Killers-list*. The former records the processors to be aborted on completion of the current transaction and the latter records the list of processors who are allowed to abort the current transaction. This is used to avoid false aborts.

Sun Microsystems was the first to develop a commercial processor with TM support [18, 29]. Their Rock processor comes with two new instructions `chkpt` and `commit` which specify the start and the ending of the atomic block. `Chkpt` takes an address (*fail-address*) as the argument which is the location to resume the execution in case of a failure. All the speculative stores are buffered in an on chip queue. The shared L2 cache is notified about these speculatively modified entries, which tracks reads of the other processors to detect conflicts. If this queue gets full before a transaction is committed, Rock proposes to abort the transaction. Therefore the size of the transaction's write set is limited to 32 entries, which is the size of the speculative buffer. When a transaction is aborted, the cause of the abort is stored in a register called *Checkpoint Status* (CPS).

Azul Systems also have developed a commercial processor with TM support [25] to accelerate Java locks. As the intention is to accelerate mutually exclusive Java code regions, they do not introduce any programming language keyword to define an

atomic region, hence rely on the existing *synchronized* keyword. Three new instructions `SPECULATE`, `ABORT` and `COMMIT` have been introduced. In order to track read and write sets, caches are extended with *speculatively-read* and *speculatively-written* bits. Therefore the size of a transaction is limited by the size of the L1 cache. If a speculatively modified cache line is evicted from the cache, the transaction is aborted. Azul rely on software heuristics to determine when to use the HTM facility available within the chip.

Intel recently announced Intel Transactional Synchronization Extensions (Intel TSX) in a future processor, codenamed Haswell [55]. Intel TSX supports two modes of execution for providing optimistic concurrency: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). HLE which is basically similar to SLE [87], is for executing legacy code which is written using locks. HLE provides two new instructions, `XACQUIRE` and `XRELEASE`. When the application code tries to acquire a lock, `XACQUIRE` is used and the lock variable is added to the read set. No write operation is performed on the lock variables, thereby making it available to others. While this enables multiple threads to acquire the same lock, if any thread has performed a conflicting read/write operation, the lock region is executed again, without lock elision. RTM provides three new instruction `XBEGIN`, `XEND` and `XABORT` to start, commit and abort a transaction, respectively. Similar to Rock [18, 29], `XBEGIN` also takes an address as the argument, fallback address, to resume the execution in case of a failure. Conflicts are detected at cacheline granularity in Haswell. Therefore if the read/write set exceed the L1 cache capacity, the transaction needs to be aborted. In addition, transactions need to be aborted in events like the execution of a `CPUID` instruction or the occurrence of exceptions.

2.7 Software Transactional Memory

In Software Transactional Memory (STM) all the TM related operations (conflict detection, version management) are performed in software. This increases the flexibility of a TM system as different policies can be used depending on the scenario. Also an STM does not encounter the physical limitations such as the buffer size, faced by an HTM system. However they have the inherent disadvantage of having to perform everything in software, which in general is slower than using hardware. The work presented in this thesis is mainly focused on HTM, therefore surveying the area of STM is of lesser relevance, however for the sake of completeness a brief summary of

two proposals are presented in the remaining section. Interested readers are directed to Chapter 4 of the Transactional Memory book [42] by Harris *et al.*, for a comprehensive summary of the available STMs.

The first STM discussed, is proposed by Shavit and Touitou [98]. Their STM only supports static transactions in which all the memory locations that a transaction might access have to be known in advance. The *Memory* data structure is used to store the speculative updates. An *ownership* vector records the owner of each block in *Memory*. Another vector *Add* which is maintained per process, contains the set of address that a transaction accesses. Old values are kept in a vector called *Oldvalues* which is updated on a successful transaction. A transaction first acquires the ownership and writes the old values to the transaction's record. New values are calculated thereafter and the memory is updated with those. Ownerships are freed thereafter. If a transaction fails to acquire the ownership, it is considered a failure.

The STM proposed by Shavit and Touitou [98] has a constraint that the data set of a transaction needs to be defined in advance. This issue was addressed in the Dynamic STM (DSTM) [49] proposed by Herlihy *et al.*. In DSTM, when a transaction requires to access an object, it creates a clone of it. Thereafter all the modifications are done on the clone object. If a transaction is to commit successfully, the pointer of the object is changed to refer to the clone object. Only the transaction has a reference to the clone object, therefore no other thread will see the speculatively modified object until it is committed. When a transaction requests a clone of an object, the cloning function checks whether any other thread has already been given a clone of that object. If that is the case, the request is denied. However this may limit the parallelism because it does not consider whether the objects have been cloned for a read operation or a write operation. In order to remedy this, the authors propose to first open all objects in a read-only mode and to walk through the objects to decide which objects are going to be speculatively modified. The authors also propose a release mechanism to remove an object from its read mode in order to reduce conflicts. DSTM was an influential proposal in STMs, since then quite a number of proposal have been made and a good summary of those can be found in Chapter 4 of the Transactional Memory book [42] by Harris *et al.*.

2.8 Hybrid Transactional Memory

HTM and STM are combined in Hybrid Transactional Memory (HyTM) to achieve the benefits of both. In HyTM some of the TM related operations are performed in software whilst the rest are performed in hardware. Damron *et al.* [28] propose to use the underlying HTM to boost performance, and to retry in the STM if the execution in HTM fails due to limitations. Their design comes with a compiler and a STM library. The compiler produces two versions of the application code, one for HTM and one for STM. Initially a transaction is tried in HTM, if it fails a function in their HyTM library is called which then decides whether to retry in HTM or in STM. Their HTM system is equipped with a function to access read and write sets maintained by the STM in order to detect conflicts between transactions running in HTMs and STMs.

Concurrently with Damron *et al.*, Kumar *et al.* also propose a Hybrid Transactional Memory system [59]. Their HyTM is based on extending the DSTM (an object based STM system proposed by Herlihy *et al.* [49]) to work with HTM. DSTM relies on creating a new object on the first transactional access to it. While this gives the flexibility to commit or abort simply by resetting the pointer, the allocating space and copying data is a costly operation. In order to remedy this costly operation, Kumar's HyTM suggests to modify data objects in-place while operating in hardware mode. In addition, transactions do not need to perform a commit-time validation as in DSTM [49] because conflicts are detected eagerly using cache coherence protocols. Similar to the HyTM by Damron *et al.* [28], Kumar's HyTM also decides a mode (software or hardware) for operation at the beginning of a transaction. Initially a transaction is tried in hardware mode, if it fails it is retried in software mode.

Lev *et al.* propose Phased Transactional Memory (PhTM) [67], which executes transactions using the best available platform. PhTM has several modes of execution. When operating under `HARDWARE` mode all the transactions are executed using HTM. When operating under `SOFTWARE` mode, all the transactions are executed using STM. All the transactions are executed using the HyTM when operating under the `HYBRID` mode. PhTM also supports `SEQUENTIAL` and `SEQUENTIAL-NOABORT` modes which are basically software modes without the overhead of managing read and write sets as no conflict detection phase is involved. In addition, `SEQUENTIAL-NOABORT` does not require the logging of memory operations as no abort operation is involved in that mode. In order to ensure that the correctness is not compromised by having different modes of execution, PhTM proposes to complete all the transactions in one mode before switching to another.

SigTM, a HyTM system proposed by Minh *et al.* [75], uses hardware signatures to detect conflict in a STM. Unlike other HyTMs, there is no switching between modes of execution. All the TM related operations like versioning, committing, aborting are done in software. Signatures are updated and conflicts are detected using existing cache coherence protocols. Therefore no modifications are required to the caches to detect conflicts. Since versioning is done in software they can support unbounded transactions without aborting, unlike other HyTM systems. As conflicts are detected using cache coherence, SigTM is able to provide strong isolation as well.

Again the survey provided in this section is not comprehensive as it does not coincide with the main focus of the thesis, however interested readers are directed to Chapter 5 (Section 5.2.3) of the Transactional Memory book [42] by Harris *et al.*.

2.9 Summary

This chapter presented the both theoretical and implementation details of TM systems. As the thesis is focused on HTM, the majority of the chapter is devoted to HTM related literature. However to make the reader aware of other available implementation platforms, it also gives an overview of software and hybrid TM systems. The intention of the chapter is to provide background material to facilitate the reader to read the rest of the thesis. Separate chapters/sections describing and comparing the closely related work to each contribution are presented in the respective parts of the thesis.

Part I

DaCTM: Data Centric Transactional Memory

Chapter 3

DaCTM: Data Centric Transactional Memory

This is the first chapter of Part I of this thesis. The chapter describes the concept of Data Centric Transactional Memory. After making the case for a system supporting synchronization, coherence and consistency using the "data centric" approach in Section 3.2, the concept of DaCTM is presented in Section 3.3. Section 3.4 discusses several issues that can arise when using DaCTM approach and the proposed solutions to address them. Finally Section 3.5 summarises the chapter.

3.1 Introduction

Given the increasing rate of the number of cores per chip, parallel programming is becoming mainstream. Simply most programmers would assume "shared memory" and sequential consistency(SC) [62] as the default programming model and memory model respectively. This simplicity comes at the cost of hardware support for maintaining a global view of the shared memory. The term "global view" means that every processor is aware of the operations done on the shared memory by other processors. In order to ensure this, processors need to communicate with each other. Conventionally this is achieved via cache coherence protocols.

When providing a global view of the shared memory with conventional cache coherence protocols, the issue of the "cache coherence wall" [58] is encountered. In a system with hundreds, if not thousands, of cores the interconnect can easily be saturated with these coherence messages. Hence the approach can become impractical. The message passing programming model does not require a global view, but comes at

the cost of explicit communication at the programming level, adding extra complexity that is not imposed by the shared memory approach. Therefore, the aim is to design a scalable system without compromising the inherent advantages of the shared memory approach.

In its strictest form, to maintain a global view of the shared memory, every processor is required to see the modifications made on every memory location in the same order. In order to achieve this, a cache coherence protocol propagates a newly written value to other processors, and the memory consistency model defines when this propagation must be done. In a parallel program not all the threads are interested in all the variables accessed/modified by other threads. In terms of processors, this means that not all the processors are interested in the modifications made by others. Therefore cache coherence and memory consistency could be enforced selectively to the locations that are of interest to other processors. To achieve this, a mechanism is needed to communicate the required level of coherence and consistency of each memory location, from the high level program to the underlying architecture. Initially, this may appear as an extra burden. However several keywords (eg: `private` [82], `synchronized` [83]) have been introduced in parallel programming languages to distinguish memory accesses. The premise of this work is that an architecture can be extended to take advantage of such information provided by programmers, in maintaining the global view of shared memory and propose a new multi-core architecture which has the potential to overcome the "coherence wall".

Access pattern of a memory location can be used to define whether it requires synchronization, how easy/hard it is to reorder operations to that location with respect to others and, whether the caches need to communicate with each other to avoid using stale data. For example if the `synchronized` keyword is used in a program written in Java [83], it can be deduced that variables enclosed with that block requires synchronization. Similarly in a program written in OpenMP [82], when the keyword `private` is being used for a variable, that variable is guaranteed to be accessed only by a single thread. If the memory locations of similar access patterns can be grouped together, they can be allocated in memory regions according to their group. For example all the local variables can be allocated in one region and the all concurrently read/write variables can be allocated in another region. In a high level program a developer already knows this information and it is a matter of communicating it to the underlying hardware.

To this end, Part I of the thesis proposes DaCTM: **Da**ta **C**entric **T**ransactional **M**emory, a transactional memory [50] system coupled with the data centric concept

[106]. The premise of the work is to associate the access pattern of each memory location with the required level of Synchronization, Coherence and Consistency (later in all the chapters of Part I this is referred as SCC). In this way the global view requirement is maintained per memory location, whilst preserving the shared memory abstraction. The DaCTM approach to parallel computing has the potential to overcome the "coherence wall". The following contributions are made in the part I of this thesis.

- A mechanism to maintain coherence and consistency based on memory regions is introduced. In this approach the address space of a program can be viewed as a collection of non-overlapping memory regions, each having a predefined level of coherence and consistency. The union of all the regions is equal to the available address space.

- An application programming interface (API) to manage the memory regions, is also introduced. In this way the programmer is relieved from manually managing different memory spaces.

- As the third contribution, a proposal is made to attach scratch-pad memories (SPMs) [5] to each processor to implement one type of memory region (LO)(*see Section 3.3*). This removes the need to use the interconnect for memory accesses related to this region, thereby reducing the contention.

- Overall design of the architecture to support the above mentioned region-based coherence and consistency, is presented as the last contribution. The evaluation of DaCTM shows that with the proposed approach, bus utilization and contention, processor idle time and false transaction aborts can be greatly reduced thereby aiding scalability. The performance evaluation presents improvements of up to 4.52 times speed-up over an optimized baseline TM system that uses lazy versioning and lazy conflict detection (an improved TCC [39]).

3.2 Motivation

Memory consistency models define the event ordering on shared memory parallel computers. Most programmers assume sequential consistency (SC) [62] as the default memory model. The advantage of having SC is that execution of a parallel program

can be seen as an interleaving of the parallel processes/threads/instructions on a sequential processor. This enables programmers to use the simple shared memory approach to do parallel programming. Current shared memory multi-core processors use cache coherence protocols that rely on small low latency messages to keep a coherent view of the memory. If this approach is used in a system with hundreds or thousands of processors, the interconnect could easily be saturated because of these messages.

Transactional Memory Coherence and Consistency (TCC) [39] proposed to maintain coherence at bulk level thereby eliminating these low latency coherence messages. In addition, the authors also showed that one simple protocol can be used to maintain synchronization, coherence and consistency (SCC). In their approach, a coherent global view is maintained at block level. The proposal is to operate atomically and speculatively within a block and to communicate the modifications at the end. Their approach was attractive in terms of eliminating the conventional coherence messages that are incurred due to write operations performed by processors. A TCC processor does not produce coherence messages for each write operation performed within a block, instead all the modifications are communicated at the end of the block. This approach demands a higher bandwidth. In a parallel program not all the memory locations are accessed by all the threads (for example `private` variables in OpenMP [82]). Therefore communicating the changes made to these sort of variables incurs unnecessary overhead, which is reflected as the high bandwidth requirement in TCC. This can be avoided if the hardware is made aware of the required levels of SCC of each memory location. In a high level program this information can be found by looking at the access pattern of the variables, from which the required levels of SCC can be determined.

```
long sum = 0, loc_sum = 0;
#pragma omp parallel for private(w,loc_sum)
{
  for(i = 0; i < N; i++)
    {
      w = i*i;
      loc_sum = loc_sum + w*a[i];
    }
  #pragma omp critical
  sum = sum + loc_sum;
}
printf("\n %li",sum);
```

Figure 3.1: Example use of the *private* keyword in OpenMP

Parallel programming languages have introduced different keywords to distinguish different types of data. OpenMP [82] has the keyword `private` to denote that a particular memory location is local to that thread. In the example code, shown in Figure 3.1, written in OpenMP, the `private` clause has been used for variables `w` and `loc_sum`. The information that can be extracted from this code is that variables `w` and `loc_sum` are local to each thread. Therefore any write operation to these locations does not require to be communicated to other processors. Simply, no coherence is required for `w` and `loc_sum`.

Java [83] has the keyword `final` to emphasise that the data is immutable. In addition, it also has immutable data structures like `String` which never gets modified. Consider the Java code segment shown in Figure 3.2. There, the keyword `final` has been used when initializing variables `radius`, `xpos`, `ypos`, `zpos`. This indicates that these variables never gets modified during the lifetime of the program. The information that can be extracted from this is, that coherence is not required for these locations. For example in a cache coherence protocol, the cache controller is not required to maintain the state bits for these memory locations. Also operations to these locations can be reordered as required since the only operation that can be made on these locations is "Read".

```
public class Sphere {
    public final double radius=2;
    public final double xpos=3;
    public final double ypos=3;
    public final double zpos=3;
    …
}
```

Figure 3.2: Example use of the *final* keyword in Java

Most of the parallel programming languages support locks (eg: `pthread_mutex_lock`, `synchronized`) to maintain synchronization. Consider the Java code segment shown in Figure 3.3. Several points can be extracted from this code. The first and the obvious one is that operations within the block needs to be atomic and mutually exclusive. The second observation is that the processor which performs the write operations on `lastName` and `nameCount` variables must send a message to other processors to invalidate their local copies for these memory locations. The third observation is that if there is any read operation to the `lastName` variable after the `synchronized` block, it cannot be issued before the write operation to the same variable takes place.

Programmers place barriers in programs to ensure that all subsequent accesses wait

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Figure 3.3: Example use of the *synchronized* keyword in Java

until all the previous accesses are completed. This is encountered with data that is produced at a given time and consumed subsequently in the program order. Applications that are developed according to the *producer-consumer* model falls into this category. Consider a multi-threaded environment for the code segment shown in Figure 3.4. In such a situation several threads can be producing whilst others are consuming. It cannot be guaranteed that an `item` will be consumed by the same thread that produced it. Therefore write operations to a particular `item` need to be communicated to other processors before any read operation to that `item` happens in the `consume` method. Even though these `item` variables are shared among threads, no synchronization is required among accesses to the same `item` by different threads, as they do not occur concurrently. The information that can be extracted from this code segment is that: coherence is required for these memory locations; any read operation to an `item` needs to be delayed until the write operation is globally performed by the producer of it; synchronization is not required among accesses to the same `item`.

```
procedure producer() {
    while (true) {
        item = produceItem();
        …
        putItemIntoBuffer(item);
        …
    }
}

procedure consumer() {
    while (true) {
        …
        item = removeItemFromBuffer();
        …
        consumeItem(item);
    }
}
```

Figure 3.4: A pseudocode of a Producer-Consumer application

From the examples discussed so far, it can be seen that required levels of SCC for most of the memory locations can be extracted from the access patterns of those locations or using the explicit keywords used in the program. If this information is communicated to the architecture, it can select different hardware operations. The crux of the proposal is to associate the required level of SCC of a memory location with the memory location itself, which is the basic principal of the "data centric" approach to programming. In the light of this, the proposal is to group together data structures or individual memory locations (later in the discussion the term object is being used to refer to either of them) having similar access patterns and to allocate them in different memory regions. The aim is to trigger different hardware operations based on the region of a particular object. Since the intention of the proposal is to handle SCC as a whole, an improved version of TCC [39] is chosen as the baseline architecture. DaCTM proposes to couple this improved TCC with the data centric concept to deliver a computing model which has the potential to overcome the "coherence wall".

3.3 DaCTM Concept

The concept of DaCTM is based on deriving the SCC of objects from their access patterns. As the first step, a *type* field is associated with each object and in the current version this *type* remains immutable throughout its lifetime. DaCTM supports the following *types*.

3.3.1 Local (LO)

These objects are accessed only by the owner processor. Since only one processor accesses these objects, neither synchronization nor coherence is required. Regarding consistency, any sort of reordering/bypassing can be allowed as long as a write followed by a read, as well as a read followed by a write to the same location by the same processor respects the program order. Consider the code segment shown in Figure 3.5, taken from Lee-TM [108].

There the function `connect` is executed in parallel and each thread allocates a 2D array of size `GRID_SIZE X GRID_SIZE`. This array is only being read and written by the thread who created it. Therefore this array can be declared as LO, hence can be allocated in the *local* memory region of the processor.

```
void *connect(void *parameter)
{
   int **tempg = (int**)malloc(GRID_SIZE*sizeof(int*));
   tempg[0] = (int*)malloc(GRID_SIZE*GRID_SIZE*sizeof(int));
   for(j=1; j<GRID_SIZE; j++)
   {
      tempg[j] = tempg[j-1] + GRID_SIZE;
   }
   …
   …
   …
   free(tempg[0]);
   free(tempg);
}
```

Figure 3.5: A memory allocation request that can be considered as LO

3.3.2 Read Only (RO)

Objects that never get modified throughout their lifetime belong to this category. They can be read by more than one processor. Since these objects do not change their value, they do not require coherence or synchronization. Since the only operation that can be performed on this type of objects is a "read", they can be reordered as needed. Consider the code segment shown in Figure 3.6, taken from Barnes application of the SPLASH benchmark suite [109].

```
static long Child_Sequence[NUM_DIRECTIONS][NSUB] =
{
 { 2, 5, 6, 1, 0, 3, 4, 7},  /* BRC_FUC */
 { 2, 5, 6, 1, 0, 7, 4, 3},  /* BRC_FRA */
 { 1, 6, 5, 2, 3, 0, 7, 4},  /* BRA_FDA */
 { 1, 6, 5, 2, 3, 4, 7, 0},  /* BRA_FRC */
  …
  …
 { 3, 0, 7, 4, 5, 6, 1, 2},  /* FDA_BDC */
 { 3, 0, 7, 4, 5, 2, 1, 6},  /* FDA_BLA */
};
```

Figure 3.6: A memory allocation request that can be considered as RO

In the application this array is read in the `find_my_bodies` function which is executed in parallel. Since this array never gets modified, it can be declared as RO, hence can be allocated in the *read-only* memory region.

3.3.3 Concurrently Read and Write (CRW)

This category corresponds to objects that can be read and written concurrently by different processors. Since they are accessed concurrently, synchronization has to be

maintained among accesses. Since more than one processor accesses the same object, reads and writes from different processors must respect the global program order. Coherence is required to communicate the changes made by one processor to the other processors. Consider the code segment shown in Figure 3.7, which is taken from Lee-TM [108].

```
void backtrack(int **tempg, int xs, int ys, int xg, int yg)
{
   …
   while ((tx != xs) || (ty != ys))
   {
      …
      …
      global_grid[tx][ty] = OCC;
   }
   global_grid[xg][yg] = END;
   global_grid[xs][ys] = END;
}
```

Figure 3.7: A memory allocation request that can be considered as CRW

In the application, the *backtrack* method is executed in parallel and each thread writes to the same `global_grid`. Since threads are concurrently reading/writing from/to this `global_grid` it needs to be declared as CRW, hence can be allocated in the *concurrently-read-write* memory region.

3.3.4 Write Now Read Later (WNRL)

Objects which are written at a particular time and read subsequently by different processors belong to this category. For example imagine a situation where several threads are waiting on a barrier to perform a read operation on an array which is being currently written by another thread. In this situation the array is shared, but read and write operations to this array are happening at different points in the program order. As read and write operations to the same memory location is taking place at different places, the program does not require to obtain mutually exclusive access before performing an operation as required for CRW data. However coherence is needed as accesses can be from more than one processor. Considering the event ordering, a read operation to an object needs to be delayed until the previous write to the same location has been performed and vice versa. Consider the code segment shown in Figure 3.8, taken from the Ssca2 application of the STAMP benchmark suite [74].

In the application, the function `computeGraph` is executed in parallel. In the code segment shown in Figure 3.8, each thread performs several write operations to the

```
void
computeGraph (void* argPtr)
{
   for (i = i_start; i < i_stop; i++) {
      …
      …
      Gptr->outDegree[i]++;
      …
      GPtr->outDegree[i] = Gptr->outDegree[i]+1;
      …
      }
      thread_barrier_wait();
      prefix_sums(GPtr->outVertexIndex, GPtr->outDegree, Gptr->numVertices);
   …
}
```

Figure 3.8: A memory allocation request that can be considered as WNRL

`outDegree` array. After that they wait on a barrier. Once each thread finished their operations, each thread calls the `prefix_sums` function, which takes `outDegree` array as an argument. When each thread is operating inside the `prefix_sums` function, the system needs to guarantee that write operations that are performed by each thread on the `outDegree` array have been made visible to all the threads. However no synchronization is required for previous write operations because each thread operates on a completely disjoint portion of the array.

Having described the types of memory locations used in DaCTM, lets see the global view requirement for each of them. In the case of LO type, it does not require the global view property to be maintained as each location is only accessed by the creator of them. In the case of RO type, these locations can be accessed by more than one processor/thread. However as the only operation involved with them is "read", no extra effort is required to maintain the global view property. Considering the WNRL and CRW, both of them require the global view property to be maintained, but the degree to which it is required differs.

In the case of WNRL objects, they need to maintain a global view but not synchronization. These objects are written at a time (t_0) by one processor and read at a time (t_n) by another processor where $t_0 < t_n$. In order to maintain the global view, changes made to a particular WNRL object at time t_0 need to be communicated to the other processors at time t_n. Instead of doing this DaCTM proposes to update the global copy of the corresponding WNRL object and to discard any local copies before enabling the operation at time (t_n). The protocol is very similar to *reconcile* and *flush* in

Dag-consistency [8] and also to the *color_step* in the data coloring [16] programming model.

Regarding the CRW objects they need to maintain a global view and also synchronization among accesses. The primary requirement to maintain a global view is to communicate the newly modified memory locations to other processors. In its very basic form, transactions in the TM context keep all the modifications made during the atomic execution in a speculative state and communicate them to the other processors at the end. Since communicating the changes made by one processor to the others is the basic requirement to maintain a global view, TM does it implicitly. Therefore in DaCTM, TM is used to maintain the global view of CRW objects in addition to maintaining synchronization.

According to the above description, the descending order of the degree of the global view is CRW $>$ WNRL $>$ RO = LO.

3.3.5 Object Operation

Associating a *type* with an object can be done manually using the keyword used in modern programming languages (eg: `private`, `critical` in OpenMP, `final` in Java etc.). Literature in the area of escape analysis [65, 66, 93, 94] shows that in certain cases it is possible to categorise data as *local* or *shared* and this information is mainly used in memory management. This means it is also possible to use such techniques to categorise data into types used in DaCTM[1]. Regardless of the method used, all the objects in a DaCTM application has a type associated with it. In the current version of DaCTM the programmer classification of objects is trusted. In future, a classification checking tool similar to SharC [4] should be employed to ensure that the declared runtime usage for objects is correct.

The *type* information is used to decide the memory region from which the space is going to be allocated for each object. This is because DaCTM uses different memory regions to allocate objects according to their *type*. Each processor has its own memory space to allocate LO type objects. Only the owner of a particular LO memory space is allowed to allocate/deallocate from that space. In addition there exists three separate shared memory regions to allocate RO, CRW, WNRL type objects. Any processor is allowed to allocate memory from these regions. Depending upon the *type* of an object, space is allocated in one of the memory regions shown in Figure 3.9.

[1]developing the compiler support is outside the scope of this thesis, only referenced to show the possibility of using it in a future version of DaCTM

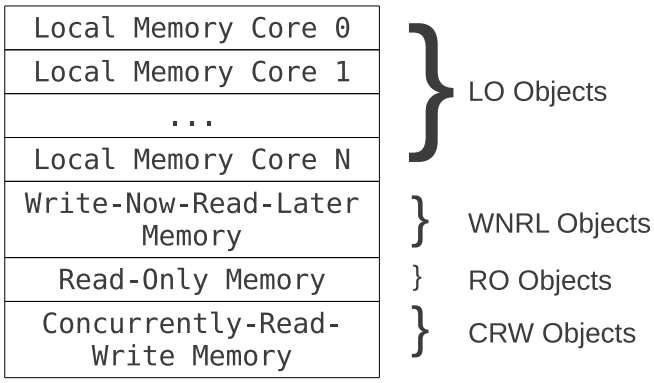

Figure 3.9: DaCTM memory regions

The memory allocation function is modified in DaCTM in order to accept an extra argument, which is the *type*. If the *type* field is not present, memory is allocated from the CRW memory region. The advantage of doing so is that, if the programmer is not certain about which type to use for a certain object, the *type* filed can be left blank and the space will be allocated from the CRW region, which guarantees the highest degree of SCC. In this manner DaCTM ensures that the correctness is not compromised even if the type information is not available. The Lee-TM [108] code segment shown in Figure 3.5, modified to allocate memory from the LO region is shown in Figure 3.10. There, the memory allocation function takes an extra argument which is the type of the object.

```
void *connect(void *parameter)
{
   int **tempg = (int**)MALLOC(LO, GRID_SIZE*sizeof(int*));
   tempg[0] = (int*)MALLOC(LO, GRID_SIZE*GRID_SIZE*sizeof(int));
   …
   free(tempg[0]);
   free(tempg);
}
```

Figure 3.10: Proposed memory allocation function in DaCTM

The proposal of DaCTM is to maintain SCC at bulk level than at individual memory locations as in conventional protocols, hence a unit to measure a block needs to be established. In this regard DaCTM considers a method body as a unit of work as in the first proposal to use Data Centric Synchronization (DSC) [106]. That said, programmers have the flexibility to break a method into several sub work units or to combine several methods to one work unit. This is described later in the chapter.

Operations are required to associate with methods in order to provide SCC for the data that the method operates on. Since DaCTM has already categorised objects, the

type information can be extended to methods as well. Therefore methods can also be characterised either as LO, RO, WNRL or CRW according to the data they operate on. If a method contains only LO objects, then it becomes a *LO* method and the same continues for RO, CRW and WNRL objects. When a method has mix of objects type is determined according to the relation CRW $>$ WNRL $>$ RO = LO. This is shown in Table 3.1.

Data types	Allocated Type
RO, LO	LO
WNRL, RO, LO	WNRL
CRW and any	CRW

Table 3.1: Determining the *type* of a method when operating on mix of data

If a method contains only LO or RO type objects, it is categorised as LO. However there is no difference even it is categorised as RO since no explicit operation is involved while executing it. A CRW method is executed as a transaction [50]. Also a WNRL method, is executed as a transaction. That said, at the architectural level, the way in which the Atomicity, Consistency and Isolation (ACI) properties are maintained for CRW and WNRL objects differs. Therefore both WNRL and CRW methods may appear as transactions at the high level language, but this does not apply at hardware level. Proposed operations for each type is shown in Table 3.2.

Data types	Operation
LO	none
WNRL	*transaction*
CRW	*transaction*

Table 3.2: Operations to perform for each data type

Classifying a method to be of a certain type is done statically. If there is not enough information available to do this, DaCTM requires them to be of type CRW, thereby requiring them to be executed as transactions. This produces a correct output because there is no harm in executing a method as a transaction even if it is not required.

There might be situations where a larger method has an already defined small transaction in the middle of the code. In such scenarios, DaCTM takes the first half of the method (that is from the beginning of it to the existing transaction) as one unit and defines the required operation. Thereafter it takes the remaining portion of the method

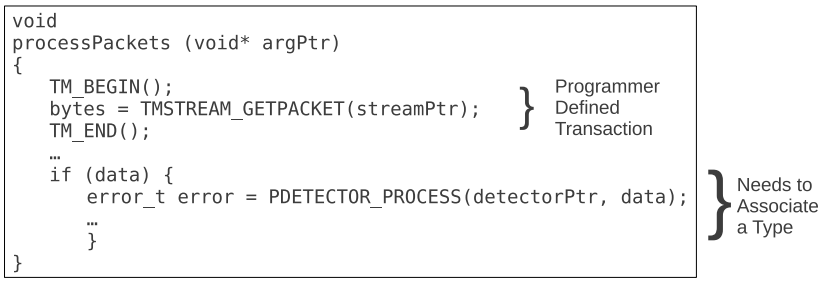

Figure 3.11: Working with explicitly defined transactions in DaCTM

as another unit and defines the required operation. For example consider the code segment shown in Figure 3.11, taken from Intruder application of the STAMP benchmark suite [74]. There, inside the `processPackets` method, there is an already defined transaction. That transaction does not encompass the entire method. Therefore an operation needs to be associated with the rest of the method in order to maintain SCC for those memory locations. This is done by associating a *type* to that portion of the method and selecting an operation according to Table 3.2.

```
void stepsystem(struct local_memory *my_Local, long ProcessId){
    …
    maketree(my_Local, ProcessId);
    …
}
void maketree(struct local_memory *my_Local, long ProcessId){
    …
    my_Local->Current_Root = (nodeptr) loadtree(my_Local, p, ...);
    …
}
nodeptr loadtree(struct local_memory *my_Local, bodyptr p, ...){
    …
    *qptr = (nodeptr) SubdivideLeaf(my_Local, le, ...);
    …
}
cellptr SubdivideLeaf(struct local_memory *my_Local, ...){
    …
    c = InitCell(my_Local, parent, ProcessId);
    …
}
cellptr InitCell(struct local_memory *my_Local, cellptr parent, ...){
    …
    c = makecell(my_Local, ProcessId);
    …
}
cellptr makecell(struct local_memory *my_Local, long ProcessId){
    …
}
```

Figure 3.12: A Chain of functions taken from Barnes application of SPLASH [109]

Another situation that needs to be considered is when an application has a chain of function calls. Consider the code segment shown in Figure 3.12, taken from the Barnes application of the SPLASH benchmark suite [109]. There, function `stepsystem`

calls `maketree` which in turn calls `loadtree` and this chain continues until function `InitCell` calls function `makecell`. As the first step, a *type* is associated with each function in the chain. If more than one of them requires to be executed as transactions, *i.e.* CRW or WNRL, special consideration is required. Lets assume all the functions in the chain are of CRW. DaCTM proposes two solutions for this kind of scenario. One approach is to use nested transactions as shown in Figure 3.13. The other approach is to start a transaction at the beginning of each method and to commit it before calling the next method. This is shown in Figure 3.14.

```
void stepsystem(struct local_memory *my_Local, long ProcessId){
    TM_BEGIN();
    …
    maketree(my_Local, ProcessId);
    …
    TM_END();
}
void maketree(struct local_memory *my_Local, long ProcessId){
    TM_BEGIN();
    …
    my_Local->Current_Root = (nodeptr) loadtree(my_Local, p, ...);
    …
    TM_END();
}
nodeptr loadtree(struct local_memory *my_Local, bodyptr p, ...){
    TM_BEGIN();
    …
    *qptr = (nodeptr) SubdivideLeaf(my_Local, le, ...);
    …
    TM_END();
}
cellptr SubdivideLeaf(struct local_memory *my_Local, ...){
    TM_BEGIN();
    …
    c = InitCell(my_Local, parent, ProcessId);
    …
    TM_END();
}
cellptr InitCell(struct local_memory *my_Local, cellptr parent, ...){
    TM_BEGIN();
    …
    c = makecell(my_Local, ProcessId);
    …
    TM_END();
}
cellptr makecell(struct local_memory *my_Local, long ProcessId){
    TM_BEGIN();
    …
    TM_END();
}
```

Figure 3.13: Nested transactions for chain functions in DaCTM

Even though DaCTM considers a method as a unit of work, the programmer always has the flexibility to break a method into several work units or to combine several methods into one work unit by using explicit transactions. This is because if atomicity and isolation are required to be maintained across methods, then programmers can explicitly use `TM_BEGIN` and `TM_END` to mark the region of the transaction. For example,

```
void stepsystem(struct local_memory *my_Local, long ProcessId){
    TM_BEGIN();
    …
    TM_END();
    maketree(my_Local, ProcessId);
    TM_BEGIN();
    …
    TM_END();
}
void maketree(struct local_memory *my_Local, long ProcessId){
    TM_BEGIN();
    …
    TM_END();
    my_Local->Current_Root = (nodeptr) loadtree(my_Local, p, ...);
    TM_BEGIN();
    …
    TM_END();
}
nodeptr loadtree(struct local_memory *my_Local, bodyptr p, ...){
    TM_BEGIN();
    …
    TM_END();
    *qptr = (nodeptr) SubdivideLeaf(my_Local, le, ...);
    TM_BEGIN();
    …
    TM_END();
}
cellptr SubdivideLeaf(struct local_memory *my_Local, ...){
    TM_BEGIN();
    …
    TM_END();
    c = InitCell(my_Local, parent, ProcessId);
    TM_BEGIN();
    …
    TM_END();
}
cellptr InitCell(struct local_memory *my_Local, cellptr parent, ...){
    TM_BEGIN();
    …
    TM_END();
    c = makecell(my_Local, ProcessId);
    TM_BEGIN();
    …
    TM_END();
}
cellptr makecell(struct local_memory *my_Local, long ProcessId){
    TM_BEGIN();
    …
    TM_END();
}
```

Figure 3.14: Committing before starting another function when operating with chain functions in DaCTM

consider the pseudocode of Lee-TM [108] shown in Figure 3.15. According to the previous discussion, the `expand` and `backtrack` methods would have been executed as two separate transactions. In that case, atomicity and isolation would not be maintained between the two methods, and this would produce erroneous output according to the algorithm being used. Therefore as shown in Figure 3.15, the programmer can use explicitly defined transactions to break the method level granularity in DaCTM. In such situations, data centric type derivation of methods will not take place.

It is also necessary to consider the following situation: A method is being used

```
TM_BEGIN();
expand();
if(path_found)
   backtrack();
TM_END();
```

Figure 3.15: Pseudocode of the Lee-TM [108] application

more than once in a program. In the first instance it operates on LO objects, but in the second instance it operates on CRW objects. The situation could also apply to a library function. In both situations a *type* has to be associated with the function at compile time. In both situations DaCTM proposes to execute the method as a transaction. Even though the method appear as a transaction at high level, at architectural level, the way in which the ACI properties of each transaction are maintained differs substantially. For example consider a situation where a method has been categorised as CRW, but it actually operates on LO objects. In this case, none of the ACI properties are maintained for the LO object.

3.4 DaCTM Special Cases

This section discusses the issues that might arise when following the DaCTM approach to computing. The first concern that might arise is about the number of *types* proposed in the current version. Even though the current version only supports four regions (LO, RO, WNRL, CRW), if a need arises, more regions can be introduced. If it turns out that the performance benefit is negligible for certain regions, they can even be merged.

The size of the regions and their overflow management mechanism is another issue that needs addressing. Even though the issue of handling overflows is important, it was not a priority at this stage. The solution proposed in the current version, in case of overflow, is to allocate from the CRW region. This is because, two fixed sized regions from the virtual memory are being reserved for WNRL and RO regions and CRW uses the rest of the space. When allocated from the CRW region, even for LO, RO and WNRL objects, SCC properties will be maintained. Even though this is not required, this will produce the correct output.

Other issues that need to be considered are possible TM inconsistencies and allocating memory inside library functions. TM inconsistencies might occur due to the following: (1) operating non-speculatively on LO objects inside a transaction; (2) allowing transactional cache overflows of WNRL objects. In both cases, if the first

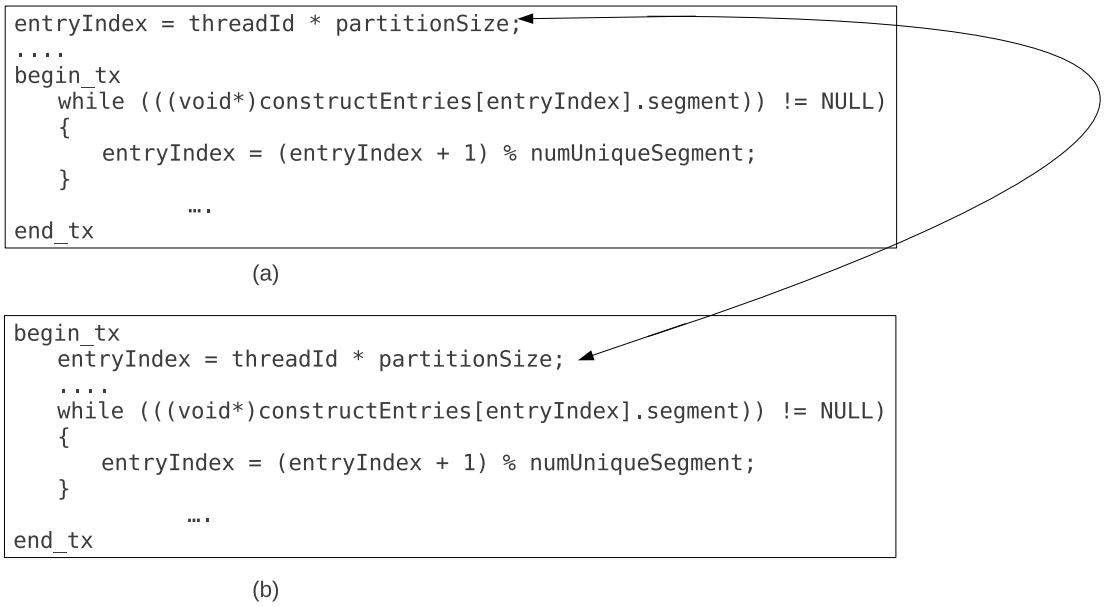

Figure 3.16: DaCTM approach to avoid the violation of TM semantics

operation is a read and, an abort happens after a subsequent write, when the transaction is restarted, it would read the new (but wrong) value. Sanyal *et al.* [96] report that only 1% of the accesses in the STAMP [74] benchmark suite fall into this category and they propose an undo buffer to store the old value of such local variables. Their approach requires extra hardware cost and also the size of this buffer cannot be determined accurately in advance. Therefore, in DaCTM this issue is addressed through the programming model. DaCTM proposes to move the initialization of LO objects inside the transaction. Consider the code segment shown in Figure 3.16 which is taken from the Genome application of the STAMP [74] benchmark suite. There, Figure 3.16(a) shows the original code in which variable `entryIndex` is initialised outside the transaction. Since no other threads are interested in the changes made to `entryIndex`, it can be categorised as LO. If the transaction gets aborted after one or more iterations in the while loop, the value of the `entryIndex` would have changed from its original value. When the transaction is restarted, a new (but wrong) value is read for `entryIndex` during the first read operation. As proposed above, in the code shown in Figure 3.16(b), the initialization of `entryIndex` has been placed inside the transaction. Therefore even if the transaction is aborted, the variable is reinitialized during the next transaction.

However there are situations where this solution cannot be applied. For example consider the code segment for copying a vector, shown in Figure 3.17, which is taken

```
bool_t Pvector_copy (vector_t* dstVectorPtr, vector_t* srcVectorPtr)
{
    long dstCapacity = dstVectorPtr->capacity;
    long srcSize = srcVectorPtr->size;
    if (dstCapacity < srcSize)
    {
        long srcCapacity = srcVectorPtr->capacity;
        void** elements = (void**)MALLOC(srcCapacity * sizeof(void*));
        FREE(dstVectorPtr->elements);
        dstVectorPtr->elements = elements;
        dstVectorPtr->capacity = srcCapacity;
    }
    memcpy(dstVectorPtr->elements, srcVectorPtr->elements, (srcSize * sizeof(void*)));
    dstVectorPtr->size = srcSize;
    return TRUE;
}
```

Figure 3.17: A library function used in copying vectors

from the STAMP suite. Here the object `dstVectorPtr` is read first and updated later in the function. Imagine if this method is called within a transaction with a LO object as an argument, all the operations on the LO object will be in-place. If the transaction is aborted after a modification is made to the LO object, original values have been lost hence an erroneous output is produced. The previous suggestion cannot be applied in this situation because the initialization of the vector might have taken place at the very beginning of the program and this operation might be taking place at any time in the program. In such cases, objects are categorised as CRW and then the underlying hardware will enforce TM properties. However, this does not mean all the objects stored in the vector need to be of type CRW. For example in Figure 3.17, space allocated for `elements` can be of LO or RO or WNRL or CRW. The only requirement is the vector object to be of type CRW to ensure the correctness of the computation.

```
bool_t Pvector_copy (int type, vector_t* dstVectorPtr, vector_t* srcVectorPtr)
{
    long dstCapacity = dstVectorPtr->capacity;
    long srcSize = srcVectorPtr->size;
    if (dstCapacity < srcSize)
    {
        long srcCapacity = srcVectorPtr->capacity;
        void** elements = (void**)MALLOC(type, srcCapacity * sizeof(void*));
        ….
    }
    ….
}
```

Figure 3.18: DaCTM approach to allocate memory inside a library function

Another issue to consider in DaCTM is memory allocation taking place inside a library function. DaCTM proposes to address this by introducing an extra argument to the library function which is the *type* of the object being allocated. Figure 3.18

shows a library function (`Pvector_copy`) modified to work with DaCTM. There, the *type* argument defines the type of the object that is going to be pushed to the vector, which is then passed to the memory allocation function. Also note that `free` does not take any extra argument. This is because the type is determined from the address of the object so that the appropriate memory space is freed accordingly.

3.5 Summary

This chapter presented the concept of DaCTM. Taking examples from several programming languages it showed that already programmers are explicitly distinguishing memory locations according to their access patterns. The chapter identifies four different types of access patterns and propose to associate this type with the required level of SCC of that memory location. It shows the presence of these four types in the code segments taken from known benchmark suites and suggest to group memory locations of similar types together. Thereafter it shows how to associate required operations for each method as in a “data centric” approach. Finally it also showed special situations where the DaCTM approach cannot be applied and also proposed solutions to overcome them.

Chapter 4

Architectural Support for DaCTM

This chapter describes how to extend an architecture to support the DaCTM concept. First, in Section 4.1, it discusses a naive way to support DaCTM concept in hardware. Later in the same section it also explains why such a design cannot be used in certain scenarios. Thereafter the design of DaCTM is presented in several sections. It is composed of DaCTM Support for Memory Region (Section 4.2), DaCTM support for Transactional Memory (Section 4.3) and two versions of DaCTM systems proposed to address the issue of transactional cache overflow (DaCTM-CS (Section 4.5) and DaCTM-U (Section 4.6)). The section also shows how it can overcome the difficulties that a naive design cannot handle. Finally Section 4.7 summarises the chapter.

4.1 Naive Design

A program written for the DaCTM architecture can be seen as a collection of transactions and non-transactions. The fact that a transaction exists in the code does not necessarily mean that the entire block is going to be executed atomically and in isolation. In DaCTM architecture the definition of a transaction changes according to the *type* of the data it operates on. Objects of a DaCTM application have a *type* associated with them. This *type* defines what sort of operations are required to maintain SCC of each object. Since DaCTM relies on conveying the *type* of an object to the underlying architecture, in a naive design it is only required to have different types of instructions for different types of data. This can be done using extra instructions similar to those shown in Table 4.1. In this approach when an application code is translated to machine code, all the conventional read and write operations can be replaced with the appropriate one. For example if the operation is a “Read” and the memory location is of type

LO, then a READ_L instruction can be used. Based on the type of the instructions being executed, hardware can perform required operations to maintain SCC. For example if WRITE_WNRL is executed, hardware needs to perform a coherence operation. Similarly if WRITE_L is executed, no operation is required to maintain coherence. As described here, this approach is less complex and requires less modifications to existing hardware. This design works when the application program is small.

Data types	Naive Instructions
LO	READ_L, WRITE_L
RO	READ_O
WNRL	READ_WNRL, WRITE_WNRL
CRW	READ_CRW, WRITE_CRW

Table 4.1: Instructions to be used in a naive DaCTM design

However there are certain situations where this approach cannot be used. Imagine the case of an application in which a particular function is being used at different places. For example consider the code segment shown in Figure 4.1. There, a pointer to an array is passed to the function sum and it sums up the content of the array and returns the result. It is very likely that this function will be reused to calculate the sum of different arrays.

```
int sum(int* arg)
{
     int total=0;
     for(int i=0;i<length;i++)
     {
          total+=arg[i];
     }
     return total;
}
```

Figure 4.1: A code segment for totalling an array

When this code is translated to machine code, for the read operation related to the variable arg, the compiler has to decide which of the read operations in Table 4.1 to use. There is no information available during the compilation time to make this decision. When the sum function is used in an application, there can be situations in which the type of the argument arg is either LO, RO, WNRL or CRW. Since the function sum is already being translated to machine instructions, different hardware behaviours depending on the *type* of the argument cannot be performed. If the compiler inserts the READ_CRW instruction, then the architecture will try to ensure that coherence

is maintained for `arg` even if it is of *type* LO. On the other hand if the compiler inserts the `READ_LO` instruction, then the architecture will not maintain coherence even if the *type* of `arg` is CRW.

A quick solution might be to have four versions machine code for `sum`. That is one version with all LO instructions and other with CRW instructions and so on. It would work in this scenario, however if the number of arguments in the library function increases, 4^R functions are needed where R is the number of arguments. This is because it is not guaranteed that all the arguments will be of same type. This requires compiled versions for all the combinations. Therefore DaCTM does not follow this approach as its architecture.

4.2 DaCTM support for Memory Regions.

DaCTM associates the required levels of SCC of a memory location with the memory location itself. Objects are allocated in the corresponding memory region (shown in Chapter 3, Figure 3.9) according to their *type*. Then the responsibility of maintaining the required level of SSC of the regions is passed to the DaCTM architecture. In order to support region based coherence and consistency, an architecture should have either separate physical memories or a logical partitioning of the shared memory for these regions. DaCTM follows a mix of both. It uses separate physical memories for LO memory regions. For RO, WNRL and CRW memory regions it uses a logical partitioning of the shared memory.

4.2.1 LO Memory

DaCTM propose to attach a separate physical memory to each processor to act as the LO memory of that processor. LO type objects are only accessed by the processor that allocates them and in doing so, there is no interconnect usage. For this private memory attached to each processor, DaCTM considers Scratch-pad memories (SPM) as a good candidate because of their low power and area utilization [5]. These SPMs are managed locally. In a conventional memory hierarchy, SPMs are placed at the same level as level 1 (L1) caches. Therefore in DaCTM no LO objects are cached in L1 and are only stored in the SPM of the corresponding processor.

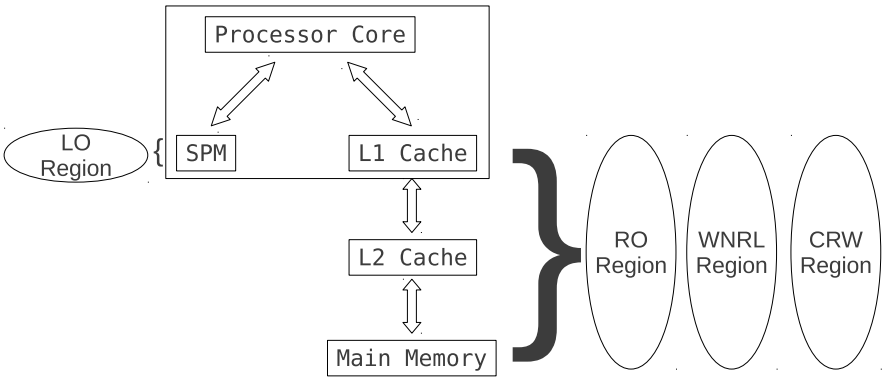

Figure 4.2: DaCTM memory hierarchy and mapping of memory regions

4.2.2 RO, WNRL and CRW Memory

All the other memory regions (RO, WNRL and CRW) use the existing shared memory. Separate regions are created for each *type* in the virtual memory. These regions are mutually disjoint and have page level granularity. The existing shared memory is used for these regions as they can be accessed by all the processors in the system. Once a program is loaded to the memory, it notifies the hardware about these different regions. Since virtual addresses are fixed for each process, these region boundaries will remain constant throughout the lifetime of an application. Figure 4.2 shows the memory hierarchy of DaCTM with the proposed SPM attached to a processor. The same figure also shows how memory regions are mapped on to this hierarchy.

When an application is loaded, it will use these regions based on the intended usage of data. An obvious question to ponder here is how an application with shared libraries use these regions. Again the answer depends on the intended usage of the data produced by these libraries. For example consider the `sum` function shown in Figure 4.1, which can be considered as a library function. In this situation, if the array which is being summed is accessed concurrently by multiple threads, then it should be allocated in CRW region. On the other hand if the array is only accessed by a single thread, it can be allocated in the LO region.

4.2.3 Region Information Table

Each processor has a Region Information Table (RIT), like the one shown in Figure 4.3, to store the boundaries of memory regions. Ideally these RITs should be maintained per process. In order to make the discussion simple, it is assumed that the number of

Type	Start	End
RO		
WNRL		
CRW		

Figure 4.3: Proposed Region Information Table (RIT) in DaCTM

available threads are equal to the number of processors. It can be observed that, in the RIT, there is no entry for the LO type. This is because region information is required only for objects that can be accessed by more than one processor. Since separate physical memories are being used for the LO region, none of the memory operations destined for them go through the conventional cache hierarchy. RIT is only required to distinguish memory locations that are stored in local caches.

4.2.4 Modifed Translation Lookaside Buffer

The Translation Lookaside Buffer (TLB) is modified to store the *type* of each virtual memory location. In the case of a TLB miss, RIT is accessed to determine the type of the incoming memory location. In case of a TLB hit, the *type* is readily available. Figure 4.4 shows the modified TLB used in DaCTM. Each object type has a different cache behaviour. Therefore when an entry is added to the L1 cache, the *type* of the location is also stored. This saves the cache controller from continually accessing the RIT to get *type* information of a cache entry to decide the required operation.

Virtual Address	Physical Address	Type

Figure 4.4: Modified TLB used in DaCTM

4.3 DaCTM support for Transactional Memory

4.3.1 Basic TM System

In its very basic form, the transactional memory implementation in DaCTM is similar to any other lazy-lazy hardware TM system. A TM system requires two new instructions to start (`TM_BEGIN`) and to commit (`TM_END`) a transaction. When the `TM_BEGIN` instruction is executed, hardware is notified by setting a flag, that it is operating inside

a transaction. In addition, a snapshot of the registers is taken and stored in a separate register file. This is used to restore the processor state in case of an abort. When a transaction is aborted, all the speculatively modified cache entries need to be flushed and the processor is reinstated from the snapshot saved in the register file. When operating inside a transaction, in a lazy-lazy TM, all the memory operations need to be performed speculatively and in isolation. That means no write operations should be made visible to other processor until the `TM_END` instruction is executed. Each processor is required to keep track of memory locations accessed/modified during the execution of a transaction, which are communicated to others when the `TM_END` instruction is executed. In a lazy-lazy TM, an abort operation takes place in the case where two processors have accessed the same memory location and one of them is a "Write" operation and the other one is a "Read" operation. A contention management policy decides which processor to abort in such a situation. Under the lazy-lazy policy, write-write conflicts do not cause a transaction to abort [39].

During this commit phase, no other processor is allowed to use the interconnect. This step is taken to prevent an aborted processor reading stale data. Imagine a situation where the processor 2 gets aborted because it has read the location `x` and the processor 1 has modified the same location. After being aborted, if the processor 2 reads location `x` for its restarted transaction, before processor 1 finished updating it, an erroneous output is produced. Therefore no other processor is allowed to use interconnect during the commit phase.

Once the next level memory is updated with all the speculatively modified cache entries, all these entries need to be cleared. This step is taken because DaCTM does not implement any cache coherence protocol. For example imagine a situation where, the processor 1 modifies the location `x` within a transaction and updates the next level copy of `x` at the end of the atomic block, but does not flush the cached entry. Thereafter the processor 2, who is also executing a transaction, modifies the location `x` and commits its transaction. During this time the processor 1 is, either not executing a transaction or has not accessed location `x` in its current transaction. Therefore, processor 1, does not get aborted by the committing of the processor 2. Now the cached copy of location `x` in the processor 1 is stale data. Thereafter whenever the processor 1 accesses location `x`, it will read a wrong value, hence will produce an erroneous output. Therefore once a transaction is committed, all the entries that are read and written during a transaction needs to flushed from the Level 1 (L1) cache.

The baseline TM architecture uses transactions to maintain synchronization, coherence and consistency. Therefore it is conceptually similar to TCC [39]. The original TCC proposal relies on obtaining exclusive commit permission when handling transactional cache overflows. In TCC, when a processor who is executing a transaction requests to overflow, conflicts are detected using the read and write bits in the cache lines. Thereafter the overflowing transaction progresses and all the other transactions are stalled. This is because, once an entry is removed from the cache, read and write sets are no longer accurate. Therefore the overflowing transaction is not able to detect conflicts with others at the time of committing. This can impose some performance overhead. For example consider the situation shown in Figure 4.5(a). There, two transactions T1 and T2 are executed in parallel. After time t_0, the transaction T1 requests to overflow, hence conflicts are detected. T2 is not aborted as there are no conflicts. Thereafter T2 is stalled and T1 is continued. After another t_1 duration T1 commits. T2 is resumed afterwards and commits later (after t_3 time). In order to complete two transactions T1 and T2, the original TCC proposal takes $t_0+t_1+t_2$ duration.

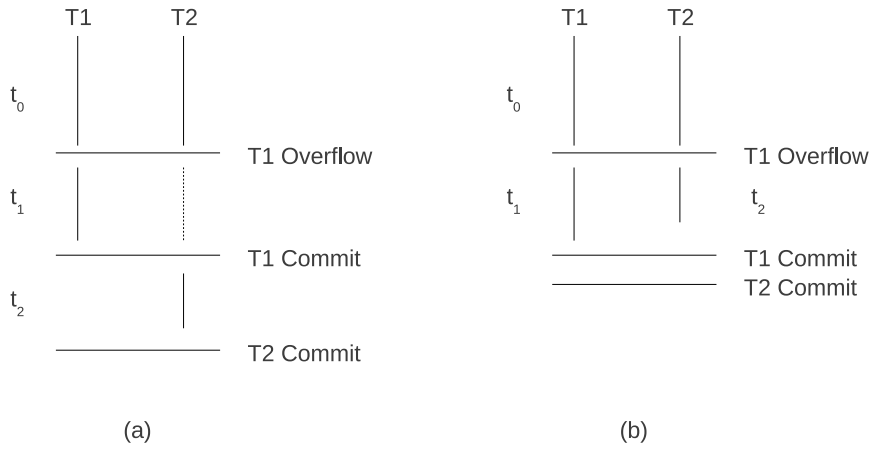

Figure 4.5: Difference between the original TCC and the improved TCC (which is used as baseline)

This performance overhead incurred by stalling transaction T2 can be addressed by decoupling conflict detection from caches by employing hardware signatures [95] (a description of signatures is presented in Section 4.3.3). With the use of hardware signatures, caches are no longer required to maintaining read and write sets. Therefore the overflowing transaction does not require to get the exclusive commit permission and to stall others. Instead all the transactions progress and conflicts are detected using signatures. This scenario is shown in Figure 4.5(b). In this situation the total time

required for completing both transactions is t_0+t_1. By employing hardware signatures the total execution time is reduced by t_2 time in the improved TCC. The commit time not considered in both situations.

If the original TCC is used as the baseline, the benefits would have come due to memory regions and also due to unnecessary serialising (Figure 4.5(a)). The aim of the experiment is to evaluate how much an architecture can be benefited from the knowledge of memory regions. Therefore, for the experiments presented in the thesis, this improved version of the TCC is used as the baseline.

4.3.2 DaCTM

DaCTM does not use any cache coherence protocols, therefore, implicitly, all the read and write operations are performed in isolation. This also applies to operations within a transaction as well. Since the TM system uses lazy versioning, no extra effort is required to operate speculatively within a transaction that fits in the L1 cache. This is because the key requirement for speculation is that the original memory location does not get modified. When a transaction is able to fit in the L1 cache, all the modifications are done on the L1 cached copy and the original value remains unmodified in the Level 2 (L2) cache or/and main memory.

When a transaction is not able to fit in the L1 cache, that is when a cache controller requests to overflow when the processor is executing within a transaction, extra operations are required to maintain the isolation property. One approach used in designing one version of the DaCTM is to allow the overflow request and to make the transaction, which the processor is currently executing, an unabortable one. The latter step, of making a transaction unabortable, is required because allowing the overflow has resulted in modifying the original value of that particular memory location. This is a non-reversible action. In order a make a transaction unabortable, none of the other processors are allowed to commit until the transaction being considered commits. Also it is worth noting here that, at any given time there can only be one unabortable transaction.

The second approach used in designing the other version of the DaCTM is to have a separate area in the memory, to which each processor can overflow their speculative modifications. In this second approach there can be any number of processors overflowing during the execution of a transaction. More description of these two systems are covered in sections 4.5 and 4.6.

4.3.3 Hardware Signatures in DaCTM

Hardware signatures [95] are used in DaCTM to keep track of all the read and write operations that are performed within a transaction. Signatures have a fixed length (eg: 1k bits [15], 2k bits [110]) and they are implemented using SRAMs. When maintaining read and write sets of a transaction, for each read and write operation, a hash function is applied to the address of the memory location. The resulting hash value is added to the signature by performing a bitwise OR operation. This is shown in Figure 4.6. There, the hash value of a given address is first created. For the sake of simplicity lets consider bit selection, which is used in Bulk [15], as the hash function. When this hash function is employed, the bit value of the address is directly fed to the hash value. Bulk [15] used following bits of the address [0-6, 9, 11, 17, 7-8, 10, 12, 13, 15-16, 18-20, 14] as the hash value. Once the hash value is produced, each bit of it is logically ORed with the existing signature, thereby delivering the resulting signature. In the proposed architecture when implementing signatures, parallel bloom filters are used in order to increase the accuracy.

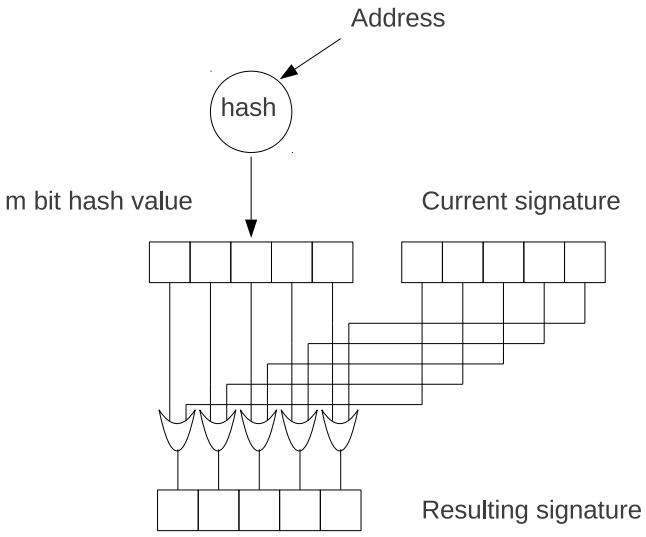

Figure 4.6: Inserting an address to a signature

When a processor needs to commit a transaction, it first requests commit permission from the centralised commit arbiter. Commit permission is granted based on a least recently granted policy. Once the commit permission is granted, the committing processor broadcasts its write-signature to all the other processors. Upon receiving this write-signature, each processor performs a bitwise AND operation with their read-signature. If all the hashes in the resulting signature are non-zero, then it is considered

as a conflict and the processor aborts. Figure 4.7 shows signature operations used in the DaCTM architecture. There, Figure 4.7(a) shows performing an AND operation between two signatures and Figure 4.7(b) shows how to check whether all the resulting hashes are zero.

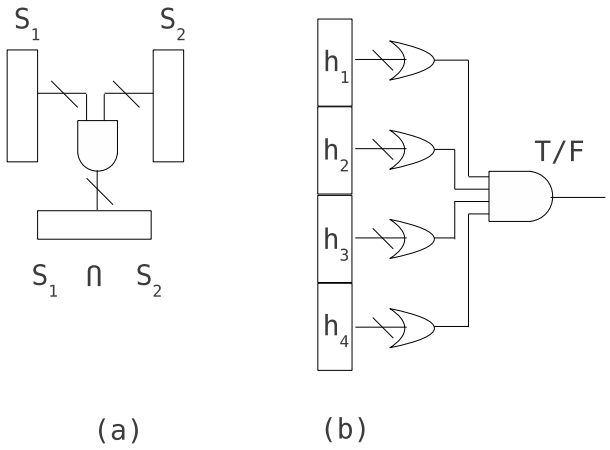

Figure 4.7: Signature operations used in DaCTM

After sending the write-signature to all the other processors, the committing processor updates the next level memory (either L2 cache or main memory) with all the speculatively modified values. During this commit phase, the communication arbiter denies any request to use the interconnect. Once the next level memory is updated with all the speculatively modified cache entries, all these entries need to be flushed and both read and write signatures need to be cleared as well.

4.4 Incorporating Memory Regions with Transactional Memory in DaCTM

Having described the baseline TM system in Section 4.3, this section discusses how to extend it to use region information. All the RO, CRW and WNRL objects are stored in L1 cache and all LO objects only stored in SPMs. Therefore the L1 cache needs to distinguish different types of memory locations. Since DaCTM does not use any cache coherence protocol, the existing entry for the state field of the protocol can be reused to store the *type* information of each cache entry. This *type* information is used for several purposes. One is to decide whether to insert the address to the corresponding signature. When a “Read” or “Write” operation takes place on a cached entry, the *type*

field is checked. In DaCTM, the address of the cache line is inserted to the relevant signature only if the entry is of type CRW.

The other usage is to decide what action to take in case of a transaction cache overflow. The baseline TM architecture, described in Section 4.3, propose either to overflow to the original memory location and to make the processor unabortable or to overflow to a separate area in the memory. With the DaCTM support for memory regions both these protocols can be improved. That is overflow permission is only sought if the overflowing entry is of *type* CRW. If it is of any other *type*, it is evicted to the original memory location. This does not pose any consistency violations because, if the location is not CRW then no other processor is interested in this memory location at this time. The last usage of the *type* information in the cache lines is to decide which entries needs to be flushed in case of an abort or a commit. In DaCTM only WNRL and CRW objects need to be flushed from L1 cache. RO objects can be kept in the L1 cache unless the space is required by an incoming cache line.

Since all the LO objects are only stored in their SPMs, the modifications made to them are made in-place. Therefore the speculative execution is implicitly dropped for all the operations performed on SPMs. Regarding the isolation, the fact that no other processors are interested in them make these objects isolated implicitly. This reduces the amount of speculative data that need to be written back to the next level memory at the end of each transaction. Regarding CRW objects, strict TM properties are maintained just as in other TM systems. Regarding RO objects, neither versioning nor conflict detection is maintained as they never get modified. Regarding WNRL objects, local copies (in L1 cache) are kept speculative implicitly. However, no extra care is taken to avoid shared copies being updated in case of a cache overflow. These objects are not considered in the conflict detection phase. If no cache overflow occurs, the behaviour of WNRL objects is similar to that of CRW objects, except that the latter objects are considered during the conflict detection phase. At the end of a transaction, the next level memory copies of both, CRW and WNRL objects are updated and local copies are flushed.

This section and Sections 4.2 and 4.3 presented the basic implementation of the DaCTM architecture. Two approaches to maintain the isolation property during transactional cache overflows are described in Section 4.5 and Section 4.6.

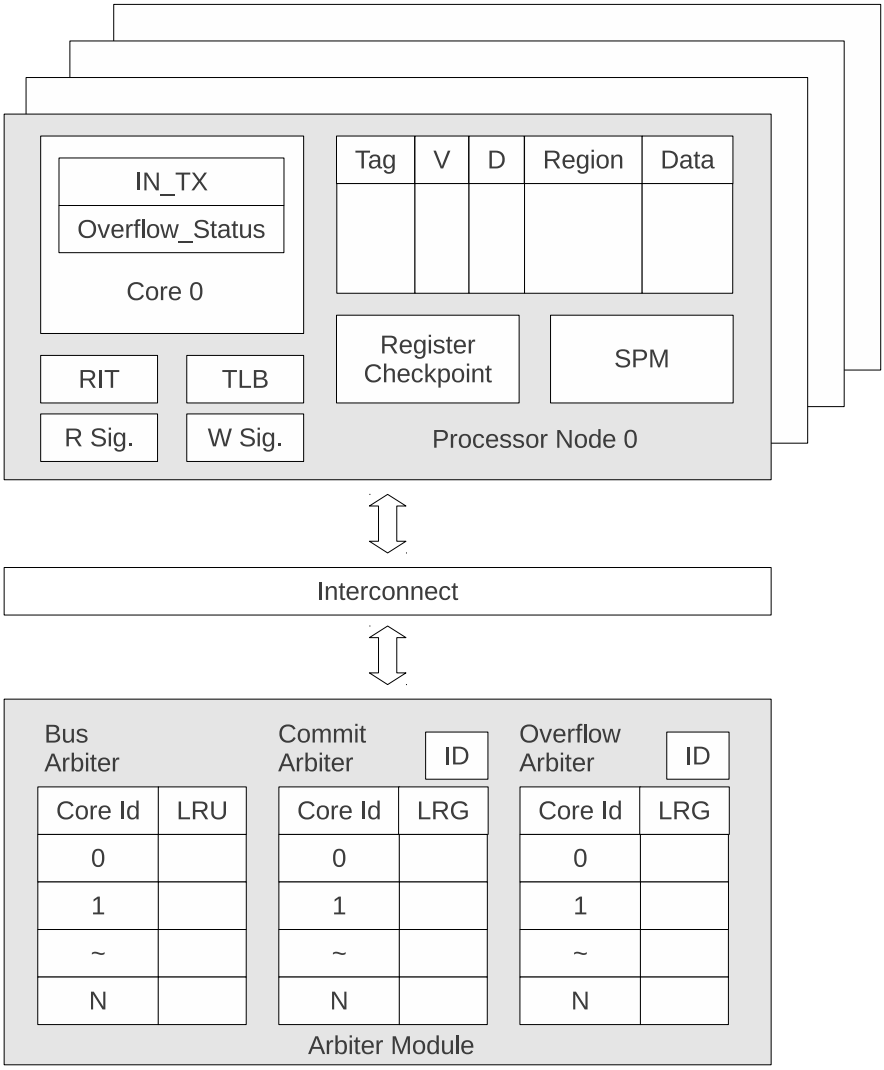

Figure 4.8: A complete DaCTM-CS system

4.5 DaCTM-CS

When addressing cache overflows, DaCTM-CS (Commit Serialize) serialises commits as proposed in OneTM-Concurrent [9]. The only difference between OneTM-Concurrent and DaCTM-CS, in terms of TM properties, is that the former detects conflicts eagerly using per block meta data, whilst latter detects conflicts lazily using hardware signatures.

In DaCTM-CS , when a cache entry needs to be rejected while a processor is inside a transaction, permission is sought from the *overflow arbiter*. Overflow permission is also granted based on a least recently granted policy. Once the overflow permission is granted, the processor flushes the cache line from its L1 cache and updates the

corresponding entry either in L2 cache or main memory. In the baseline architecture a processor is required to ask for overflow permission if a dirty entry is to be rejected while the processor is operating within a transaction. In the DaCTM-CS architecture this protocol is improved in such a way that a DaCTM-CS processor needs to ask for overflow permission only if the cache line is dirty and it is of type CRW. If it is of type RO or CRW or WNRL and the dirty bit is not set, no overflow operation is required since the value has not been modified. In the case of WNRL objects, no permission is required to overflow to the original memory location, even if the dirty bit is set. This is because, overflowing of WNRL objects to their original memory locations does not pose any consistency violation as no other processor is accessing this memory location concurrently with this processor. Therefore even within a transaction, for dirty WNRL objects, a DaCTM-CS processor is allowed to overflow to their original memory location. This does not apply to LO objects, because their modifications are made in-place in the corresponding SPMs.

If an overflow request is denied, the processor stalls until the request is granted. In this case, DaCTM-CS has an advantage over the baseline because it only asks for overflow permission if the cache line is of type CRW. In Section 4.3, the policy of the *commit-arbiter* is described as *least-recently-granted*. There is an exception to that in the CS version. That is, once the overflow permission is granted to a processor, all the commit requests from other processors are denied, until the overflowing processor commits. Again the DaCTM-CS processor has an advantage over the baseline in this situation. Since the baseline has no knowledge about the cache lines that request overflow permission within a transaction, the permission could have granted to a WNRL or LO type block. When the overflow permission is granted all the other commit requests are denied. In the case of DaCTM-CS, since it filters out cache lines, this situation would not occur. When a processor with overflow permission granted commits, it only needs to write back the remaining dirty cache entries in its L1 cache. Again in this situation DaCTM has an advantage because all the dirty WNRL objects that require overflowing have already been copied to their original memory location, where as in the baseline these entries are still waiting for overflow permission. A complete DaCTM-CS system is shown in Figure 4.8.

4.6 DaCTM-U

In order to support an unbounded amount of transactional data, DaCTM-U overflows to a separate uncached area of memory as in Large Transactional Memory (LTM) [3]. The design and the protocol is similar to that of LTM, except DaCTM-U does not stall to check for potential conflicts that might arise from overflowed locations. This is because, DaCTM-U uses signatures, and conflicts can be determined by checking signatures. When a cache line is to be evicted while operating inside a transaction, the baseline U architecture proposes to overflow to a separate area of memory. In the DaCTM-U architecture this operation takes place only if the object is of type CRW. When overflowing to this separate area of memory, the entire cache line including all the tag, valid, dirty and data bits are preserved in this uncached area.

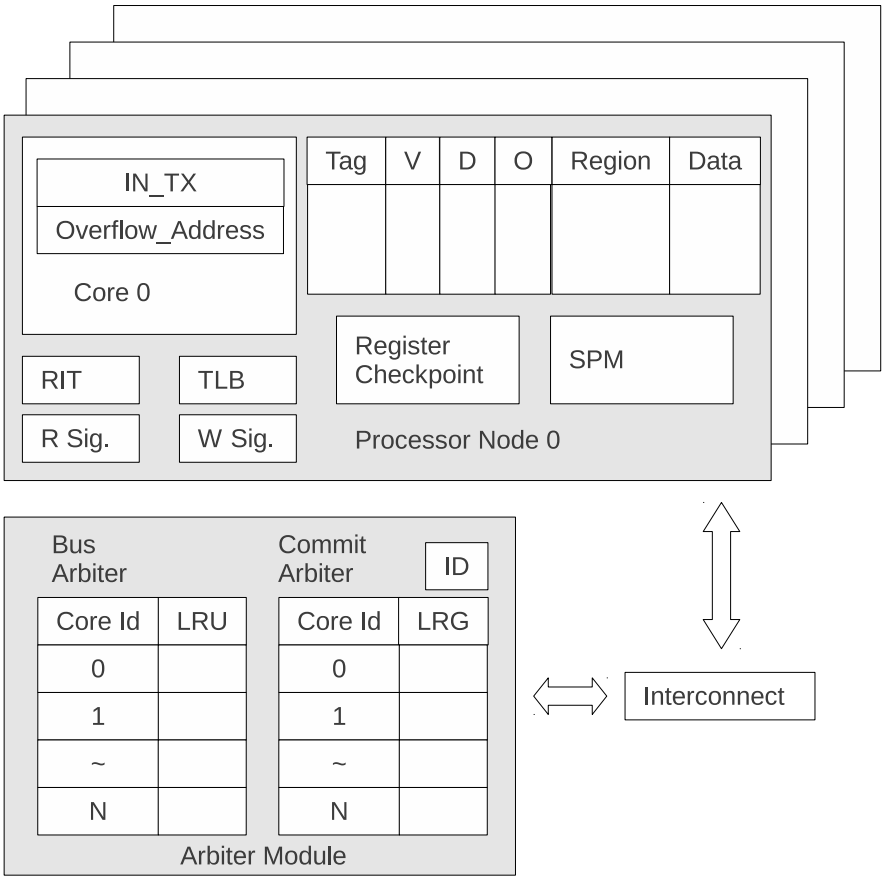

Figure 4.9: A complete DaCTM-U system

Each entry is indexed by the hash value of the overflowed memory location. Each processor has an extra register (*Overflow_Address*) which points to the starting location of this separate area. In the case where more than one memory location produces the

same index, a linked list is formed. DaCTM-U has an extra bit called O per cacheline to indicate the overflow status. This is set when a cache line is overflowed and is cleared only when a transaction commits or aborts. Even if an existing cache line is replaced with new data, this bit does not get changed. When a cache miss occurs to a cache line with O bit set, the request is directed to the overflow area of the memory. Thereafter the entry is located using the hash value of the overflowed memory location. If an entry has more than one cache line (which could happen due to the hash function mapping more than one memory location to the the same entry), a linear search is performed by comparing the tag and the index of each element in the list. If none of the entries in the current index match the memory location that caused cache miss, then it is fetched from the original memory address.

When a processor requests to commit, it first commits all the dirty entries in its L1 cache. Thereafter all the entries in the uncached area are copied to their original memory locations. In the case of DaCTM-U, it has an advantage over the baseline because, it only overflows CRW objects to this uncached area whereas the baseline architecture overflows all the objects regardless of their type. Because of that DaCTM only has to copy CRW objects from the uncached area to their original memory locations, whereas the baseline has to copy all the object to their original memory locations. A complete DaCTM-U system is shown in Figure 4.9.

4.7 Summary

The design of two architectures that are aimed to exploit the DaCTM concept is presented in this chapter. The crux of the architecture is to be aware of the types of memory locations that a program accesses. This information is stored in a hardware structure called the Region Information Table (RIT). As an optimization, Translation Lookaside Buffers have also been modified to store the *type* of a particular address. Hardware signatures are used to keep track of the read and write sets of a transaction, so that the conflict detection is decoupled from caches. In order to address the issue of transactional cache overflows DaCTM-CS proposes to serialise commits. That is to allow one processor to overflow to its original memory location and to make it unabortable. All the other processors has to wait until the unabortable one commits. The other system, *i.e.* DaCTM-U proposes to overflow to a separate area in the memory as in LTM. This version allows multiple overflowing processors.

Chapter 5

DaCTM Evaluation

This chapter describes the evaluation of DaCTM-CS and DaCTM-U. The evaluation environment is described in Section 5.1. A brief description of the benchmarks used for the evaluation and the input configurations used for each of them are presented in Section 5.2. The evaluation setup is discussed in Section 5.3. The scalability of DaCTM-CS and DaCTM-U and performance improvement of them over their baseline architectures (improved versions of TCC [39]) is presented in Section 5.4. Characterisation of the results of DaCTM on processor idle time, bus contention and false positives, is shown in Section 5.5. Finally, Section 5.6 summaries the chapter.

5.1 Evaluation Environment

Simics [70] which is a full system simulator, is used to model the both DaCTM systems. In its very basic form, any instruction executed in the simulator takes one cycle to complete. However the simulator provides two handlers namely *timing-model* and *snoop-memory*, which can be used to change the behaviour of the communication from a processor to the memory and from the memory to a processor, respectively. It also provides a feature called *haps* which are triggered when a particular event that is registered with that *hap* occurs in the simulated machine. For example a *hap* can be registered to trigger when a processor switches its mode from *user* to *supervisor* or vice versa. The advantage of this feature is that it transfers the control to a user defined function in such a situation.

Another useful feature provided by the simulator is the simulating of adding custom instructions. This is achieved by using a *magic-instruction*. This is a special *NOP* instruction and in the X86 architecture it is `xchg %bx %bx`. This *magic-instruction*

also works as a *hap*. Therefore when the `xchg %bx %bx` instruction is executed, the *magic-instruction* hap occurs and the control is transferred to a user defined function. In this manner a desired behaviour can be simulated by defining it in the user module and registering the particular user function with the *magic-instruction* hap. In order to trigger the *magic-instruction-hap*, a `xchg %bx %bx` instruction needs to be inserted in the user code.

The above mentioned functions and features are used to set-up the environment used for evaluating the DaCTM architecture. In addition to those, Simics provides quite a lot of other functions as well. Describing them all is outside the scope of this thesis.

5.2 Benchmarks Tested

This section describes the benchmarks used for the evaluation and the modifications made to them to work with an architecture that uses transactions to maintain SCC. All the benchmarks except for Lee-TM [108] are taken from the STAMP benchmark suite [74].

5.2.1 Genome

The application takes large number of gene segments and match them to reconstruct the original source genome. The `sequencer_run` method is executed in parallel and it has several user defined transactions inside the method. Therefore the other portions of the method are analysed to check which portions accesses the shared data and transactions were inserted in those places. Those were mainly the places the application code accesses data structures that are not created within the `sequencer_run` function itself.

5.2.2 Intruder

This application scans network packets and checks them against a known set of intrusion signatures. In the application the `processPackets` function is executed in parallel. The function has three user defined transactions. The rest of the function accesses data structures that has not been created within the current function, therefore that portion of the function is also enclosed with a transaction.

5.2.3 Kmeans

The application groups objects into K clusters. In the application the `work` function is executed in parallel. The function also has user defined transactions. Therefore the rest of the function is analysed to see whether it accesses any of the data structures that are not created within the current function. One such situation was found and a transaction was inserted enclosing that portion of the code.

5.2.4 Labyrinth

This is a routing algorithm similar to Lee-TM [108], which attempts to find a path form a given source point to a given destination in a three dimensional grid. In the application the `router_solve` function is executed in parallel and it has several user defined transactions. Even though the non-transactional code accesses several data structures, all of them are created within the `router_solve` function. Therefore no extra transactions have been added to Labyrinth in order to maintain SCC.

5.2.5 Ssca2

Ssca2 is an application comprising four kernels and the TM version presented in STAMP [74] focuses on kernel 1, which constructs an efficient graph data structure. In the application the `computeGraph` function is executed in parallel. The function has few user defined transactions with small transaction length. However a larger portion of the code accesses data structures that are created by another thread, mostly by the thread bearing identity zero. Therefore several extra transactions have been added to maintain SCC.

5.2.6 Vacation

This application emulates a travel reservation system. Several client threads concurrently interact with the database of the travel system. The `client_run` function is executed in parallel. The function has some user defined transactions. The majority of the parallel function is executed as transactions. The non-transactional code only accesses data structures defined within the parallel function, therefore no extra transactions have been added in this application in order to maintain SCC.

5.2.7 Lee-TM

Lee-TM is a routing algorithm proposed by Watson *et al.* [108]. The objective of the application is to find a path from a given source point to a given destination point. For the experiment, a two dimensional grid has been used. The `connect` function which is executed in parallel, has two user defined transactions. One is to find and mark routes in the shared global grid and the other is to obtain work from a list by incrementing a shared counter. An extra transaction has been added when reading the source and destination points from a global list, in order to maintain SCC.

The input configurations used for each benchmark are shown in Table 5.1. Apart from Genome-Large, all the applications from the STAMP suite use the standard input [74]. An additional input for Genome is used because, the standard input did not scale beyond 8 processors.

Application	Input
Genome	-g256 -s16 -n16384
Intruder	-a10 -l4 -n2038 -s1
Kmeans-Low	-m40 -n40 -t0.05 -i random-n2048-d16-c16.txt
Kmeans-High	-m15 -n15 -t0.05 -i random-n2048-d16-c16.txt
Labyrinth	-i random-x32-y32-z3-n96.txt
Ssca2	-s13 -i1.0 -u1.0 -l3 -p3
Vacation-Low	-n2 -q90 -u98 -r16384 -t4096
Vacation-High	-n4 -q60 -u90 -r16384 -t4096
Lee	75x75 Grid, 320 routes
Genome-Large	-g1024 -s32 -n262144

Table 5.1: Benchmark applications and their inputs used for evaluating DaCTM

5.3 Evaluation Setup

This section describes the evaluation setup used for evaluating the DaCTM architecture. First it discusses how to build a complete system in Simics as the basic configuration only comprises a processor and a memory in which all the instructions complete the execution in one cycle.

5.3.1 Building Complete System

In its very basic form the Simics simulator does not have any caches or an interconnect. Therefore the communication from processor to memory via the *timing-model* interface has been intercepted and certain delays have been inserted to simulate the cache hierarchy. In order to simulate the bus, another clocking item, slower than the processor, has been added to the system. A user module has been developed to support the coordination between interconnect, processors and the cache hierarchy. With the developed module the DaCTM system is configured with the components shown in Table 5.2. In addition to those, DaCTM-U uses a perfect hash function to index its overflowed memory locations. A perfect hash function is used, so that the performance is not affected by the quality of the hash function.

Component	Feature
Processors	1-16, in-order
L1 Data Cache	2 way assoc, 64 B line, 32 KB size, 2 cycle latency, private per core
SPM	256 KB, 2 cycle latency, private per core
Signature	2048 Bits, 4 Parallel H3 [12] Hash functions
L2 Data Cache	8 way assoc, 64 B line, 4 MB size, 20 cycle latency, shared
Interconnect	Split-transaction bus, 4 cycle latency, 64 B data width
Main Memory	100 cycle latency

Table 5.2: Components and features of the DaCTM evaluation environment

5.3.2 Building Transactional Memory Support

Since the basis of the proposal relies on transactions, a lazy-lazy hardware transactional memory system is modelled in Simics [70], a full system simulator running Linux kernel version 2.6.16. Two major requirements for implementing TM support are, to notify the hardware about start and end of a transaction and to make the hardware to operate speculatively within that region. To realise the first requirement, two extra instructions namely `TM_BEGIN` and `TM_END` were added using the *magic-instructions* feature of Simics, which allows the simulation of the addition of custom instructions. The second requirement, that is to operate speculatively, is achieved by interrupting the processor memory communication via the *timing-model* interface. Once the `TM_BEGIN` instruction is executed, the operations associated with all the subsequent instructions are buffered until the `TM_END` instruction is executed.

A snapshot of the processor registers are taken when the `TM_BEGIN` is executed and saved in an internal data structure of the user module. This is used to restore processor registers in case of an abort. The commit arbiter and the commit protocol are also implemented in the user module. The former decides to whom to grant the commit permission whilst the latter ensures all the conflicts are resolved and appropriate actions have been taken to abort the conflicting processors. It also ensures that all the next level memory copies have been updated and local caches have been flushed.

5.3.3 Support for Memory Regions

Scratch-pad memories were added as separate memory units and an unmapped address range in physical address space was assigned to these SPMs. These addresses were then mapped to virtual addresses and a separate function was added to allocate memory from these SPMs, which acted as the *local* memory region of each processor. These SPMs attached to each processor are used to allocate objects of type LO. In order to support *read-only* (RO) and *write-now-read-later* (WNRL) regions, two memory spaces were reserved from the shared memory region and two other functions were developed to allocate memory from those regions. The remaining memory space of the shared memory is used as the *concurrently-read-write* (CRW) region and the default memory allocation function is used to allocate memory from this region.

It is also required to record the memory regions in hardware. This was straightforward for the *local* region because the hardware is aware of the unmapped physical address range that was assigned to SPMs. In the case of *read-only* and *write-now-read-later* regions, this has to be explicitly communicated to the hardware. Two memory blocks of 256 MB each, are allocated from the shared memory space at the beginning of the execution of each application for these two regions. Then the hardware is notified with the upper and lower bounds of these regions. All this region information is then stored in the Region Information Table.

However, the applications used for the evaluation did not have any objects of type RO, therefore no data is allocated from the *read-only* region. Lee's routing algorithm [108] and applications from the STAMP [74] benchmark suite were used to evaluate the DaCTM architecture. Memory allocation requests of those applications were analysed manually to identify the types of object being used in the program. Thereafter existing memory allocation requests were replaced with either LO, WNRL or CRW memory requests. For comparison purposes, unmodified versions of all the applications were executed on a baseline architecture as well.

However, due to the fact that no cache coherence protocol is implemented in DaCTM, none of these applications were able to execute without being modified. Since DaCTM provides coherence using transactions, all the applications were modified by adding extra transactions in places where they access shared data. Therefore, the transactional characteristics of the applications used in the evaluation may not be similar to those presented in STAMP [74]. In the rest of the discussion, the term "unmodified applications" refers to those using default memory allocation function as in the original benchmark suite. Therefore when unmodified applications are executed on DaCTM, no region information is available, hence no filtering of objects is performed. During the rest of the discussion the term baseline is used to refer to an improved version of TCC [39] executing unmodified applications.

The reason the default TCC system is not used, is because it lacks certain features like hardware signatures to allow an unbounded amount of transactional data, as well as an uncached area in the memory to hold cache overflows like the one used in LTM [3]. The aim of experiment is to evaluate the advantages of classifying objects as of certain types and to select hardware operations based on that type. Therefore the baseline system, which is conceptually similar to TCC, has all the features a corresponding DaCTM system has except that the latter has support for memory regions.

5.3.4 Evaluation Procedure

All the evaluations are made on the parallel region of the applications. In addition to the configuration shown in Table 5.2, a baseline system with 256 KB of L1 cache was used for evaluation as well. The latency of this L1 was kept at 2 cycles which is the latency of 32 KB L1 and 256 KB SPM as well. This evaluation is done because applications which are modified to use SPM have an inherent advantage of increased level 1 storage coming from both SPM and L1 cache and the intention was to check whether the improvements in DaCTM were coming from the increased level 1 memory space or from the DaCTM support for memory regions.

5.4 Performance

Figure 5.1 shows the scalability of both CS and U versions of DaCTM. All the applications scaled well except Labyrinth and Genome. In the case of Genome, when the number of processors is increased from 8 to 16, the bus contention of the latter

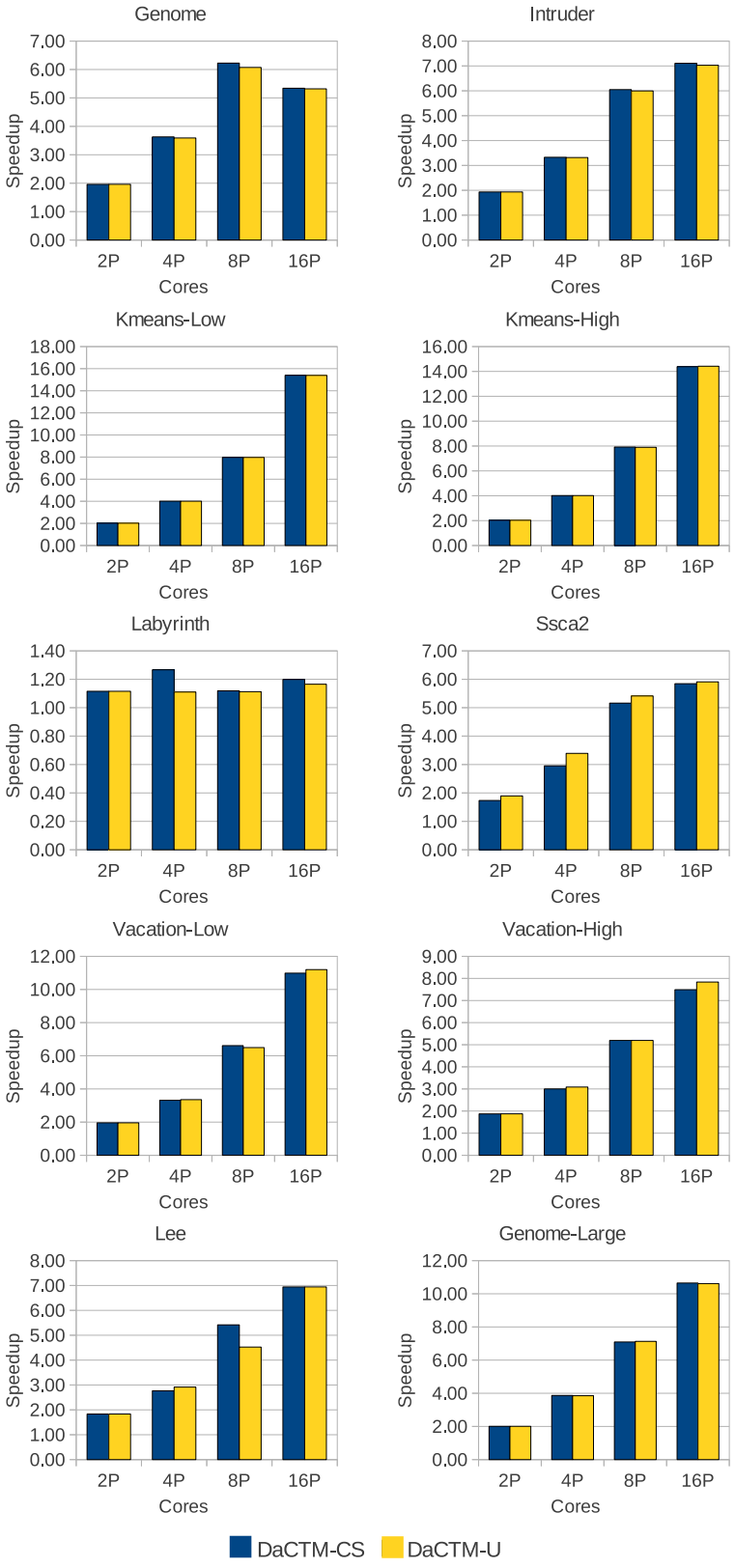

Figure 5.1: Scalability of DaCTM

becomes 2 to 3 times that of the former. This makes it fail to scale beyond 8 processors. The reason Labyrinth does not scale is because DaCTM uses signatures to keep track of the read and write sets of transactions. With signatures an element can be inserted into the set, but it cannot be removed. Therefore DaCTM does not provide the early-release [100] feature, in consequence quite a number of aborts are produced.

Among the applications, Kmeans (both Low and High) shows almost a linear speedup. The rest of the applications except Labyrinth, show a linear speedup for low processor counts (2, 4), but the trend is not continued as the number of processors increase.

Figure 5.2 shows the performance improvement of DaCTM, over baseline architectures, of all the applications used for the evaluation. In the figure, Baseline-CS represents the improvement of DaCTM-CS over the CS version of baseline architecture. Baseline-CS-256k shows the performance improvement of DaCTM (with 32 kB L1 cache) over the baseline with a bigger (256 kB) L1 cache. The same applies to Baseline-U and Baseline-U-256k. Both CS and U versions of DaCTM outperform their corresponding baseline systems, as well as the baselines with a bigger L1 cache. The performance improvement of DaCTM over baseline architectures varies from 1.06X (DaCTM-CS, Kmeans-Low) to 4.84X (DaCTM-CS, Lee).

First the discussion is focused on comparing the performance improvement of DaCTM architectures in comparison to their corresponding baseline architectures. Thereafter the effect of increasing the L1 cache of the baseline is studied. In the CS version, the highest performance improvement over the baseline is reported for Lee, which is 4.84X. From the rest of the applications, the following range of improvements are reported: Genome (1.10X $\leftrightarrow$ 2.31X), Intruder (1.20X $\leftrightarrow$ 1.28X), Kmeans-Low (1.06X $\leftrightarrow$ 1.09X), Kmeans-High (1.07X $\leftrightarrow$ 1.13X), Labyrinth (0.99X $\leftrightarrow$ 1.27X), Ssca2 (1.49X $\leftrightarrow$ 4.14X), Vacation-Low (1.40X $\leftrightarrow$ 2.82X), Vacation-High (1.58X $\leftrightarrow$ 2.89X) and Genome-Large (1.31X $\leftrightarrow$ 3.77X). When the L1 cache size is increased in the baseline, the highest improvement over the baseline has reduced to 4.52X in Lee. In the rest of the applications, the range of the improvements are as follows: Genome (1.10X $\leftrightarrow$ 1.36X), Intruder (1.20X $\leftrightarrow$ 1.27X), Kmeans-Low (1.06X $\leftrightarrow$ 1.09X), Kmeans-High (1.07X $\leftrightarrow$ 1.14X), Labyrinth (1.04X $\leftrightarrow$ 1.24X), Ssca2 (1.35X $\leftrightarrow$ 3.89X), Vacation-Low (1.38X $\leftrightarrow$ 2.70X), Vacation-High (1.54X $\leftrightarrow$ 2.70X) and Genome-Large (1.29X $\leftrightarrow$ 2.01X). Increasing the L1 cache has resulted in the following significant changes in the improvements: Genome (2.31X $\rightarrow$ 1.35X), Ssca2 (4.14X $\rightarrow$ 3.89X), Lee (4.84X $\rightarrow$ 3.31X) and Genome-Large (3.77X $\rightarrow$ 2.01X).

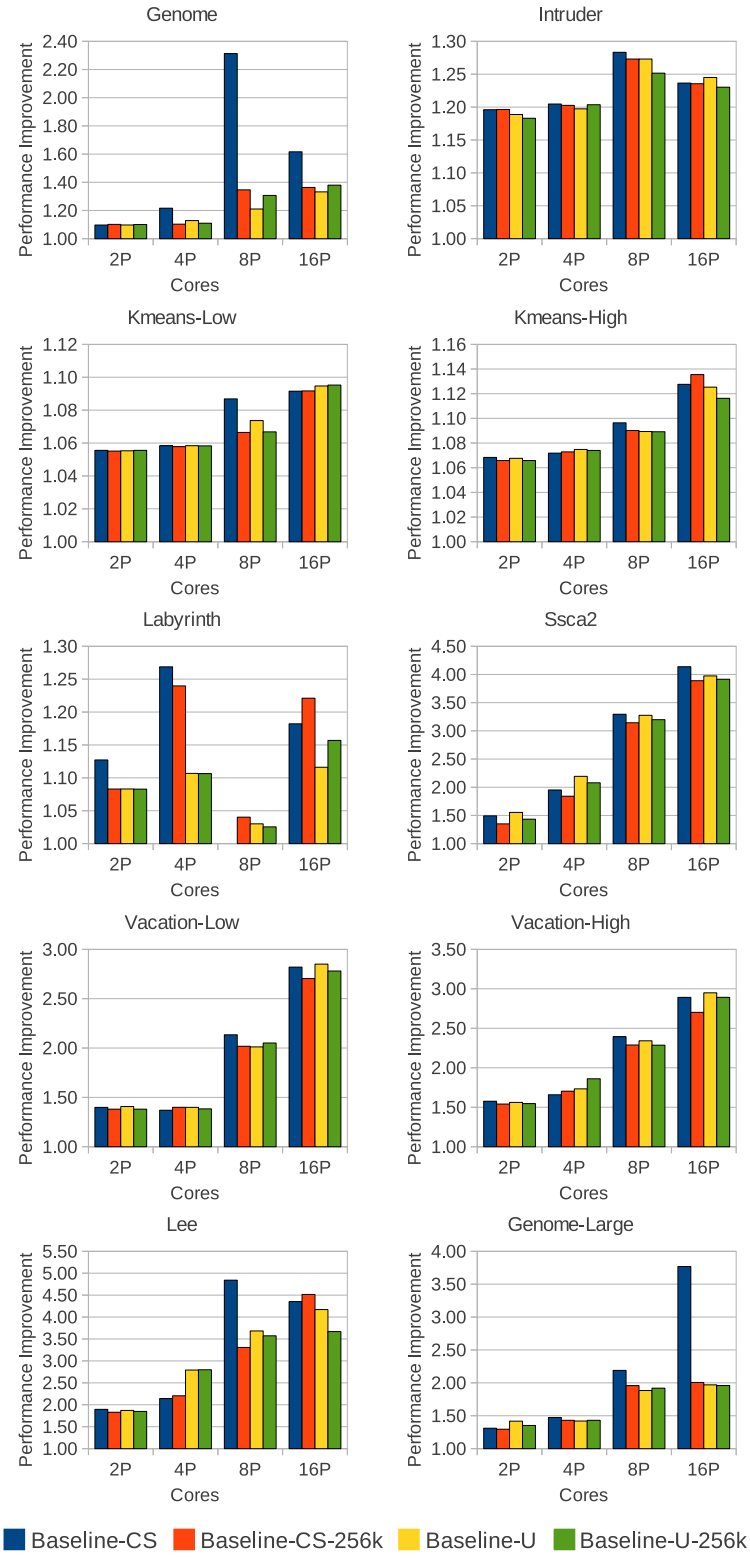

Figure 5.2: Performance improvement of DaCTM over baseline architectures

Similarly considering the U version, the highest performance improvement over the baseline is reported in Lee, which is 4.17X. The performance improvement range reported in the rest of the applications are Genome (1.10X $\leftrightarrow$ 1.32X), Intruder (1.19X$\leftrightarrow$ 1.27X), Kmeans-Low (1.06X $\leftrightarrow$ 1.09X), Kmeans-High (1.07X $\leftrightarrow$ 1.13X), Labyrinth (1.03X $\leftrightarrow$ 1.12X), Ssca2 (1.55X $\leftrightarrow$ 3.97X), Vacation-Low (1.40X $\leftrightarrow$ 2.85X), Vacation-High (1.56X $\leftrightarrow$ 2.95X) and Genome-Large (1.42X $\leftrightarrow$ 1.97X). When the L1 cache size is increased to 256k in the U baseline, the performance improvement range changes to the following: Genome (1.10X $\leftrightarrow$ 1.38X), Intruder (1.18X $\leftrightarrow$ 1.25X), Kmeans-Low (1.06X $\leftrightarrow$ 1.10X), Kmeans-High (1.07X $\leftrightarrow$ 1.12X), Labyrinth (1.03X $\leftrightarrow$ 1.16X), Ssca2 (1.43X $\leftrightarrow$ 3.92X), Vacation-Low (1.38X $\leftrightarrow$ 2.78X), Vacation-High (1.55X $\leftrightarrow$ 2.89X) and Genome-Large (1.35X $\leftrightarrow$ 1.96X). Only Lee reports a significant change in the performance improvement when the L1 cache is increased, which is 4.17X $\rightarrow$ 3.67X.

The important observation to make is that, despite the presence of certain reductions in the performance improvement for certain configurations, none of the reductions reached 1.0X. This means the performance improvement of DaCTM over the baseline is not coming from the increased level 1 storage in a DaCTM processor, which is coming from the L1 cache and the SPM. Instead it is coming from categorising objects into different *types* and triggering different hardware operations based on the *type* of the memory location concerned. More discussion on this is presented in Section 5.5.

Having observed that solely increasing the L1 cache does not reduce the execution time, the discussion is now focused on describing the causes of performance improvements in DaCTM. In the remainder of the section, the discussion is presented in very abstract manner. A comprehensive characterisation of the results is presented in Section 5.5.

In the case of Ssca2, the reduction of false aborts in DaCTM provides the main contribution for the performance improvement, this is in addition to the general contribution from the reduction in idle time and bus contention. In the case of Vacation, performance improvements are coming collectively from the reduced processor idle time and bus contention. Lee has significantly less processor idle time, bus contention and false aborts which all account for the performance improvement. The reduced processor idle time, bus contention and false aborts account for the performance improvement in Genome-Large. In the case of Kmeans, both Low and High versions report similar execution times in the baseline architectures and in DaCTM. The reason

for the low performance improvement is because this application uses small transactions and has low contention. When transactions used in the program are small, no transactional cache overflows are required. This removes the need to either serialise commits (CS) or access the uncached area of memory (U). Also when transactions are small, few signature insertions take place resulting in less false positives. For this application, two baseline configurations (2,4) also produced zero false positives.

5.5 Characterization of DaCTM

This section characterises DaCTM in terms of various parameters. Since DaCTM is based on the concept of associating a type with each object, firstly, the percentage of accesses present in each category is measured. Figure 5.3 shows the average of each access type of 2 to 16 processors for each benchmark. The majority of accesses belong to the LO type in both DaCTM-CS and DaCTM-U, from which Genome-Large shows the lowest percentage (71%) and Lee shows the highest (99%). Both Vacation High and Low, Intruder and Genome have a considerable number of CRW objects (11%, 11%, 8%, 6% respectively). The rest of the applications have less than 5% of CRW objects and in cases like Lee it is less than 1%. Regarding the WNRL objects, Ssca2 has the maximum percentage of accesses (31%). Genome-Large, Kmeans-Low, Kmeans-High, Genome also have significant number of WNRL accesses (25%, 21%, 21%, 19% respectively). Others have 5% (Intruder) or less accesses.

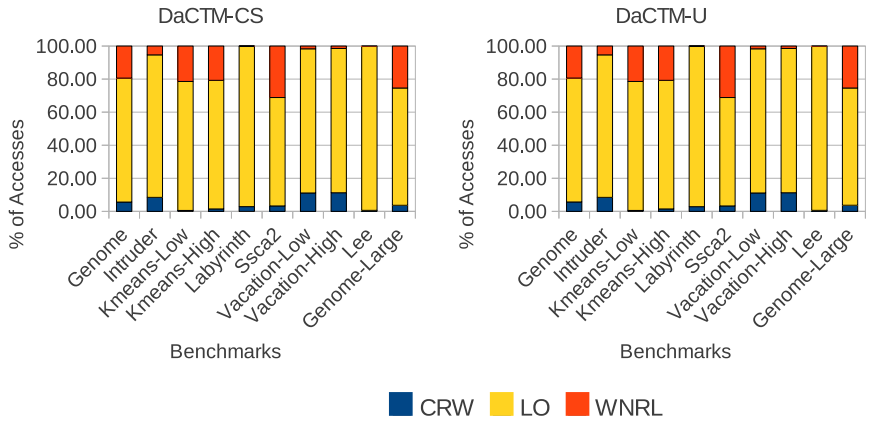

Figure 5.3: Percentage of LO, WNRL and CRW data types in both DaCTM architectures

This figure validates the hypothesis that the entire Part I of the thesis is based on, *i.e.* memory locations used in programs have different access patterns hence can

be treated differently. For example the majority of accesses belong to LO type and DaCTM proposes to allocate them on the on-chip scratch-pad memory which is closer to the processor. By doing so, none of these accesses encounter a need to use the interconnect, thereby reducing the contention for it. Also fewer CRW objects means fewer insertions to signatures, resulting less false transaction aborts. In case of a transaction cache overflow, WNRL objects can be written back to their original memory locations. In the CS version, this reduces the number of times a processor has to seek overflow permission. This in turn reduces the overflow waiting time of other processors. Overflow waiting time is the time a processor has to stall until its overflow request is granted. The request can be denied if the permission is already granted to another processor. By requesting overflow permission only for CRW objects, DaCTM reduces the overflow waiting time of processors. Also by overflowing to their original memory locations, DaCTM reduces the amount of data that needs committing at the end of a transaction.

Having seen the presence of different types of objects in the applications used for the evaluation, the discussion now focuses on analysing the impact of categorising objects into these types, with various parameters. Figures 5.4 to 5.7 show normalised values for different parameters. In each figure, Baseline-CS indicates that DaCTM-CS values are normalised to the corresponding Baseline-CS value, similarly Baseline-CS-256k indicates that DaCTM-CS values are normalised to the corresponding Baseline-CS with a 256 kB of L1 cache. The same applies for Baseline-U and Baseline-U-256k.

5.5.1 Idle Time

Figure 5.4 shows the idle time of each processor. In this experiment, a processor is considered idle if its is waiting for data to be present in its L1 cache or if its waiting for the interconnect to be available. In this regard DaCTM has an advantage because all LO objects can be stored in SPM so that they can be used without getting them from next level memory. The idle time of DaCTM is 95% (Lee) to 17% (Kmeans-Low) less when compared to their baseline systems. One could argue that this advantage is coming from the increased level 1 storage because of the added SPM in DaCTM. This argument can easily be nullified because, even with a baseline system having a L1 cache of 256 kB, processor idle time has not changed significantly in comparison to that of the DaCTM processor. In Figure 5.4, it can be seen that reduction of the idle time due to increased L1 cache, is negligible in most cases. Therefore the DaCTM approach of having a separate on-chip memory and allocating all the LO type objects

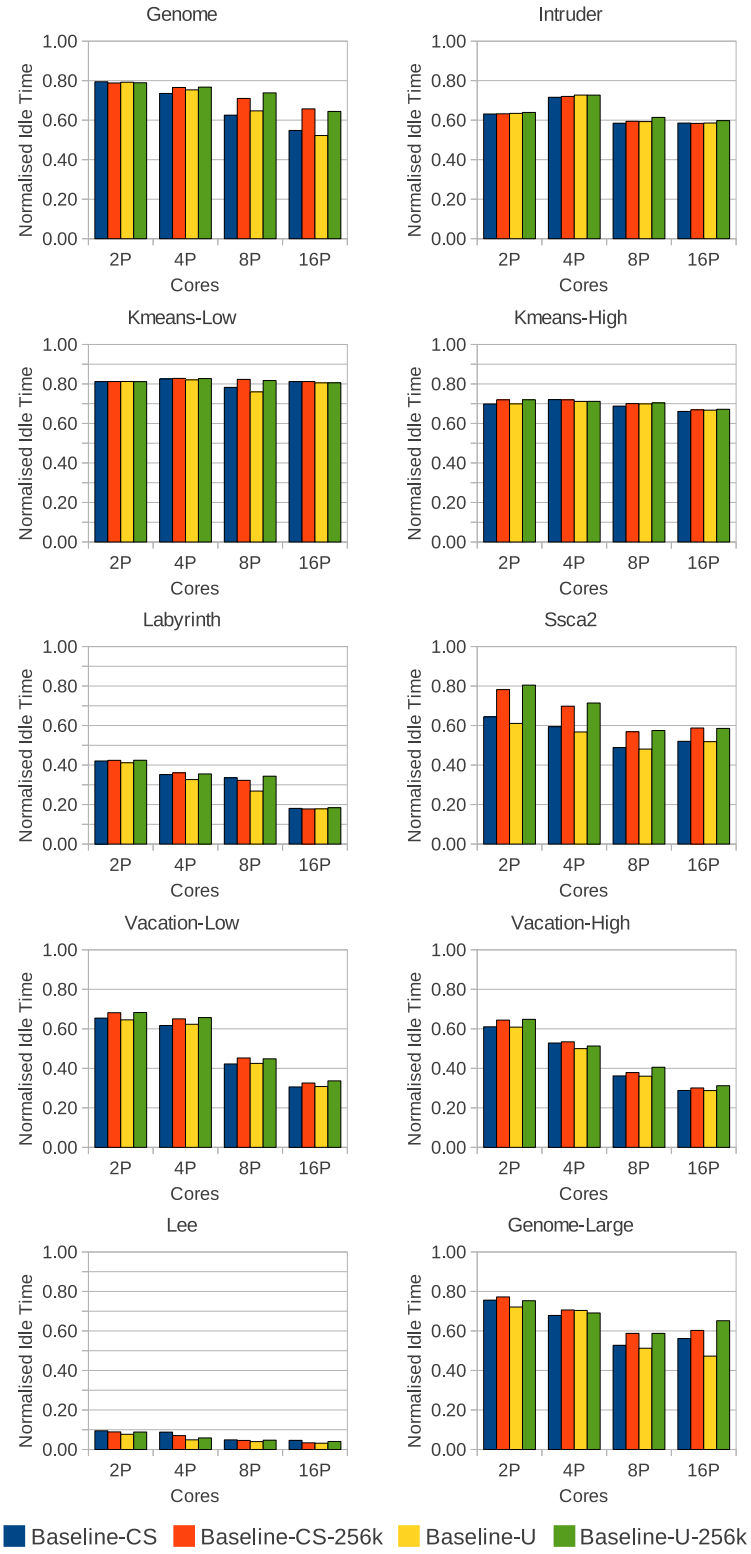

Figure 5.4: DaCTM idle time normalised to baseline

in that memory space is able to reduce the processor idle time. The evaluation also showed that increasing the existing L1 cache in the baseline architecture does not help to achieve the same impact.

In the case of Kmeans, both high and low show a similar idle time for various processor counts. This is because, it is an application with low contention and it has shorter transactions. Therefore the commit phase is kept small. When the commit phase is small, the effect of it towards the bus contention is very small. In general, when the number of processors increases, the contention for shared resources such as bus, increases. In TM, this can be aggravated if the the bus is held for longer periods during commit time. Since the commit time does not affect the bus contention in Kmeans, the processor idle time does not change as the number of processors increase. Even though the TM characteristics of Ssca2 are similar to Kmeans in the original version, in this experiment extra transactions have been inserted to the former to maintain coherence. Therefore it shows a different behaviour to Kmeans.

The rest of the applications except Intruder, show a similar behaviour. That is the normalised idle time is reduced as the number of processors increase. What happens there is that idle time of both DaCTM and the baseline architectures increases as the number of processors increase. However the rate of increase in DaCTM is low compared to that of the baseline. Therefore the normalised time reduces. Also it is worth noting that when the baseline has an increased L1 cache size, the corresponding normalised value increases. However in most cases this increase is negligible.

5.5.2 Bus Contention

Bus contention is another parameter that is relevant to the discussion. It is measured as the number of times a bus request was denied. Figure 5.5 shows these values normalised to the values reported in a corresponding baseline system. From Figure 5.5, it can be seen that all the applications got from 100% (Lee) to 54% (Ssca2) less bus request denies compared to their baseline systems. A bus request can be denied if it is granted to another processor or if another processor is committing. Therefore if other processors use the bus frequently to bring data in and out, the availability of the bus is reduced, thus contention is increased. In this particular aspect of bringing data in and out, DaCTM has an advantage because LO type data is stored in the on-chip SPM. Therefore the interconnect is not used for these type of data in DaCTM thereby reducing the contention for it. On the other hand, if processors spend more time in

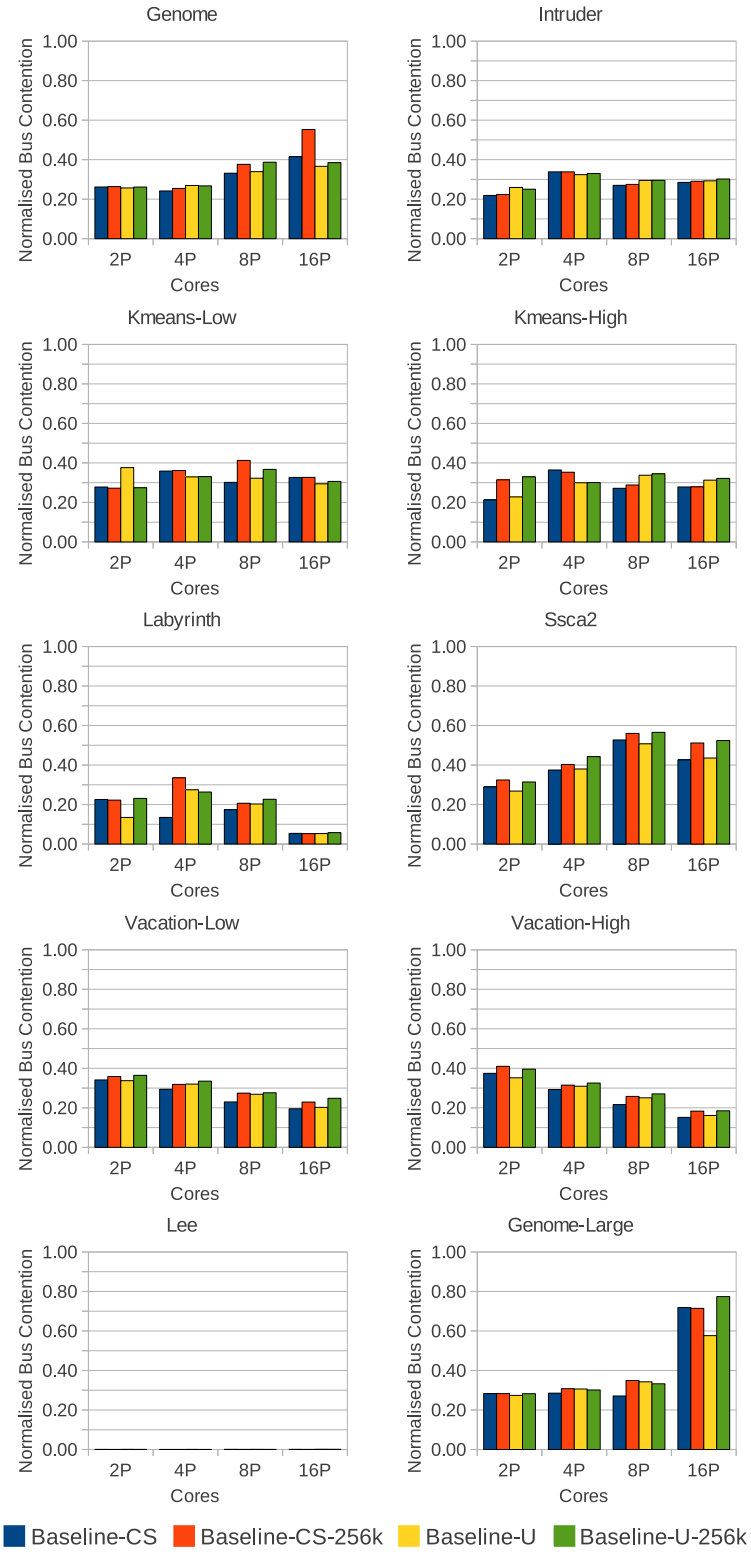

Figure 5.5: DaCTM bus contention normalised to baseline

committing, then the bus is not available for the use of other processors, thereby increasing the contention for it. DaCTM proposes to operate non-speculatively for LO objects even within a transaction. This reduces the amount of speculative data that need committing at the end of a transaction. This in turn reduces the commit time, thereby increasing the availability of the bus to be used by the rest of the processors.

In the case of both Kmeans applications, normalised bus contention does not change as the number of processors increase. The reason for the behaviour is the same as for the behaviour in processor idle time, *i.e.* being a low contention application and having shorter transactions. In Figure 5.5 two patterns can be observed. In one type the normalised bus contention increases as the number of processors increase (Genome, Ssca2, Genome-Large). In the other type normalised bus contention is reduced as the number of processors increase (Vacation-Low, Vacation-High). In general, in a multicore system contention for shared resources such as the bus, increases as the number of processors increase. Due to separation of data, DaCTM is able to reduced the usage of it. Having said that, even in a DaCTM system as the number of processors increase, contention for the bus increases. However as the DaCTM system starts the contention from a low value, there is more space to grow until the saturation point. In the case of the baseline, it started from a high value, therefore has only little space to grow. Therefore normalised bus contention increases as the number of processors increase.

Vacation behaves completely opposite to this norm because it has significantly less false positives in DaCTM than the baseline. When a transactions is aborted, its L1 cache is flushed and data is fetched again. When this happens falsely and quite a few times, it can introduce a significant amount of contention. Therefore for the Vacation applications, the baseline architectures have extra contention than those available in Genome, Ssca2 and Genome-Large. However this is not present in DaCTM. Therefore in the case of both Vacation applications, DaCTM has less contention in comparison to the baseline due to a reduction in false aborts.

5.5.3 Bus Usage

Two direct approaches used in DaCTM to reduce the bus usage are, allocating LO type objects in on-chip memory and to operate non-speculatively on those objects even within a transaction. When allocated in the on-chip memory, the bus is not used to transfer LO data to/from the processor. DaCTM proposes to operate non-speculatively on LO type objects. The advantage of this is the amount of data that needs committing is reduced. Since LO objects are allocated in the on-chip memory and operations

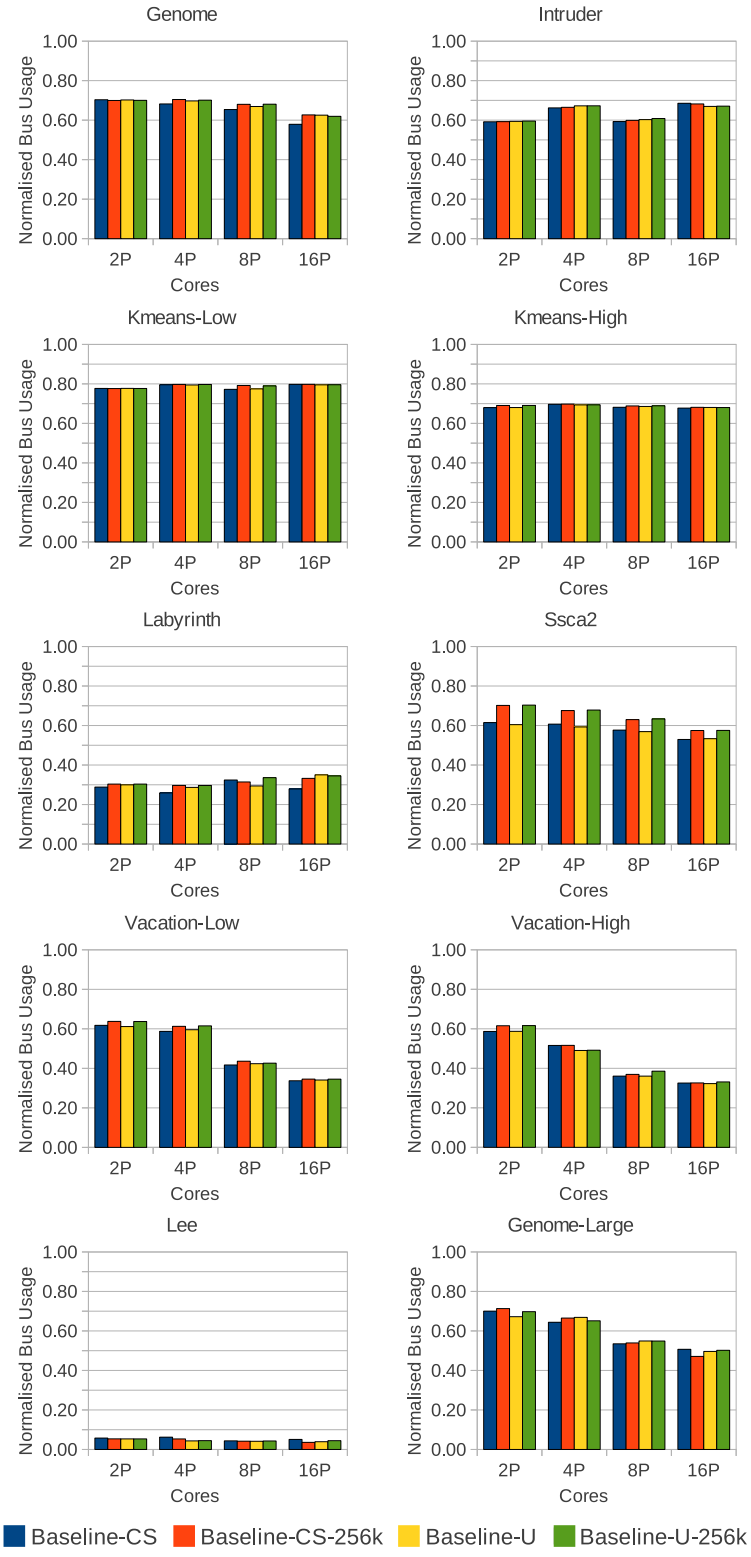

Figure 5.6: DaCTM bus usage normalised to baseline

are performed non-speculatively on them, these objects do not use the bus during the commit phase. With these two techniques, DaCTM reduces bus usage by 95% (Lee) to 20% (Kmeans). The normalised bus usage is shown in Figure 5.6.

From Figure 5.6, two patterns can be observed. One is that majority of applications (Genome, Intruder, Kmeans, Labyrinth, Lee) show similar bus usage regardless of the number of processors in the system. This is the generally expected behaviour because the work which needs to be done does not change as the number of processors increase. Therefore the amount of data that needs to be brought to the processors and the amount of data that needs to be committed does not change. Therefore bus usage is steady regardless of the number of processors in the system. However applications like Vacation, Genome-Large and Ssca2 show a different behaviour. That is the normalised bus usage reduces as the number of processors increase. This is because these applications tend to produce a large number of false positives for the baseline system as the number of processors increase. Therefore when a transaction aborts it needs to clear all its L1 cache and bring data from the next level memory, and the interconnect is used in doing so. Therefore the bus usage of the baseline increase significantly for those applications whilst it remains steady for the applications executed on DaCTM. This makes the normalised bus usage of those applications to reduce as the processor count increase.

5.5.4 Commit Phase Bus Usage

The situation described in Section 5.5.3 is clearly reflected in Figure 5.7 which shows the bus usage during the commit phase. There, it can be observed that irrespective of the processor count, bus usage during the commit time remains steady. This is because the amount of data that needs committing is independent of the number of transactions aborted due to false positives. However for the applications used for the experiment, this is dependent on the given workload. Therefore for a given application, for a given workload, the number of transactions and the amount of speculative operations needed in order to complete the given task, remains either the same or varies by a small amount. Therefore the commit time bus usage of both DaCTM and baseline architectures remains similar for all the processor configurations, hence the normalised value does not get changed. It is also worth noting that bus usage of U systems during commit time is higher than that of CS systems. This is because CS systems allow transactional cache overflows to their original memory locations, thereby implicitly reducing the number of memory locations that need committing. On the other hand, U

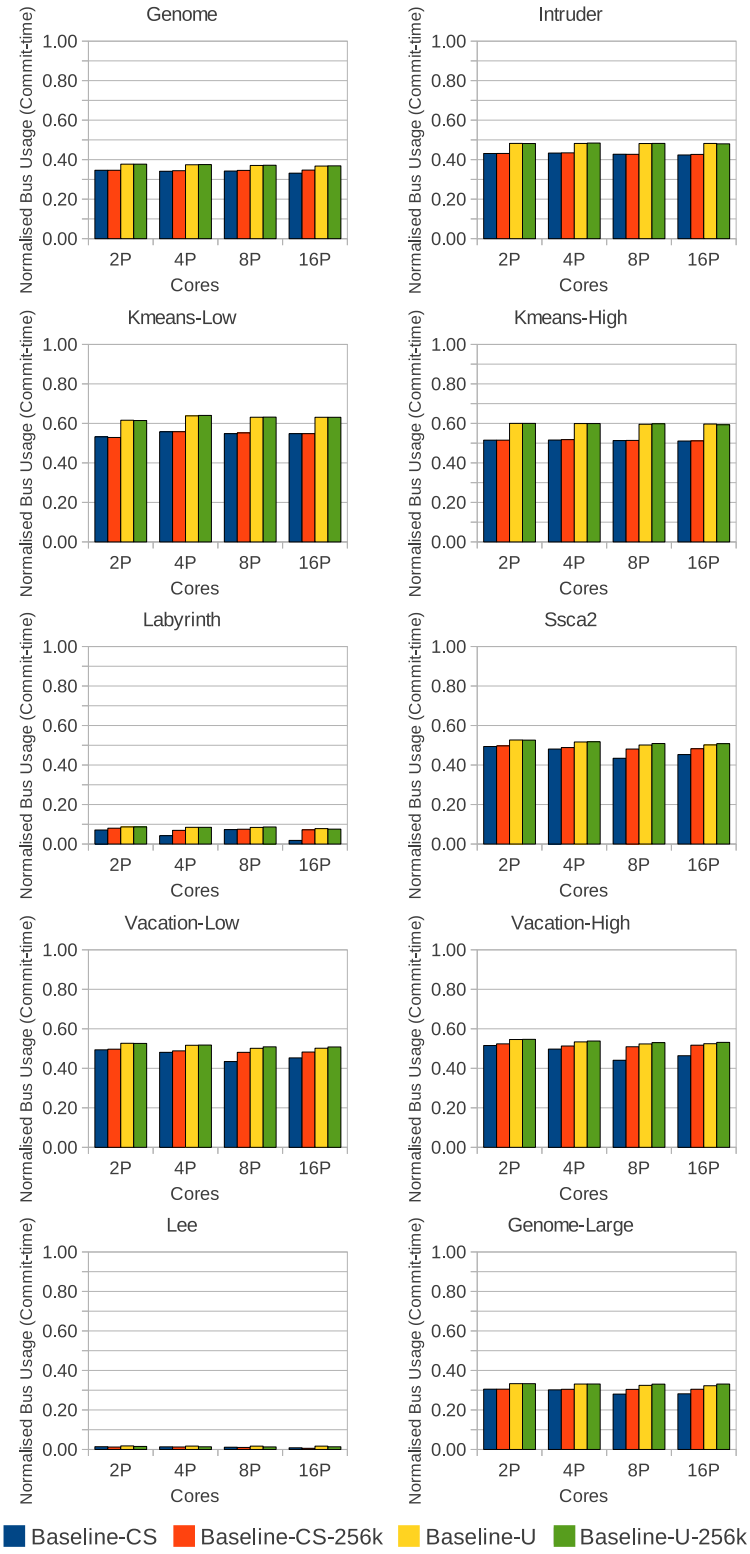

Figure 5.7: DaCTM commit phase bus usage normalised to baseline

systems allow to overflow to a separate area of memory, which need to be written back to their original locations during the time of committing. Therefore U systems utilise bus more than CS systems during commit time. Considering the CS version, the bus usage of DaCTM during the commit time is 1%(Lee) to 56%(kmeans-Low) that of the corresponding baseline. In the case of U version, bus usage of DaCTM is 2%(Lee) to 64%(Kmeans-Low) that of the corresponding baseline system.

5.5.5 Signature Insertions

DaCTM uses signatures to record the read and write sets. A notable feature of signatures is that, when detecting conflicts, they can produce false positives, but not false negatives. The more addresses being added to the signature, the bigger the probability of producing false positives [95]. DaCTM only inserts addresses of CRW objects in the signatures. Since all the applications used for the experiment have fewer CRW objects, the number of insertions made to read and write signatures are reduced. Figures 5.8 and 5.9 show the number of insertions made to read and write signatures in CS and U versions of baseline and DaCTM architectures respectively. In both figures, R-Baseline, shows the number of insertions made to the read-signature in the baseline architecture. R-Baseline-256k shows the insertions made to the read-signature of the baseline which has the increased L1 cache of 256 kB. R-DaCTM shows the insertions made to the read-signature of the DaCTM architecture. When ‘R’ is replaced with ‘W’ in the legend, the above definition is changed with write-signature.

From both the figures it can be seen that DaCTM inserts only a small number of addresses to their signatures. In the case of DaCTM-CS, Kmeans-Low (0.42%-0.43%) and Lee (0.29%-0.57%) report the lowest percentage of insertions made to the read signature in comparison to those made in the baseline. In the same architecture, insertions made to the write signature for Lee and Labyrinth are negligible in comparison to the insertions made in the baseline. Comparably higher percentages of insertions are made to the read signature in the DaCTM-CS architecture in Labyrinth (15%, 4P), Vacation-Low (13%, 2P), Vacation-High (12%, 2P) and Intruder (12%, 16P). In the case of write signatures, comparably higher percentage of insertions are made in Ssca2 (14%, 2P). The rest of the applications reported less than 5% of insertions to the write signature, in comparison to those made in the baseline. For applications like Kmeans-Low, Labyrinth, Vacation-Low, Vacation-High, Lee and Genome-Large the percentage of insertions dropped even below 1%.

Similarly in the case of the DaCTM-U architecture, Kmeans-Low (0.42%-0.43%)

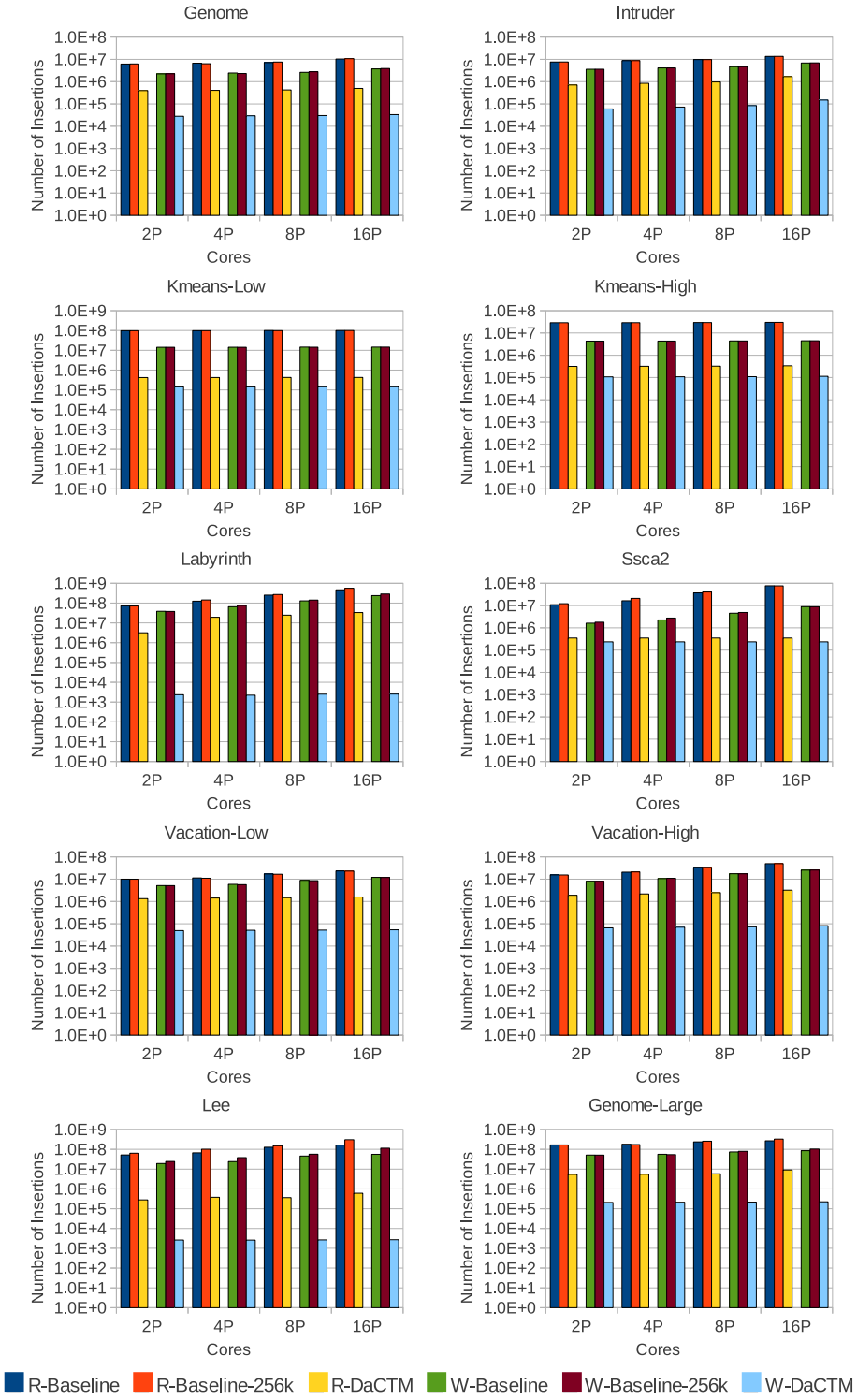

Figure 5.8: Insertions to read/write signatures in the CS version of baseline and DaCTM

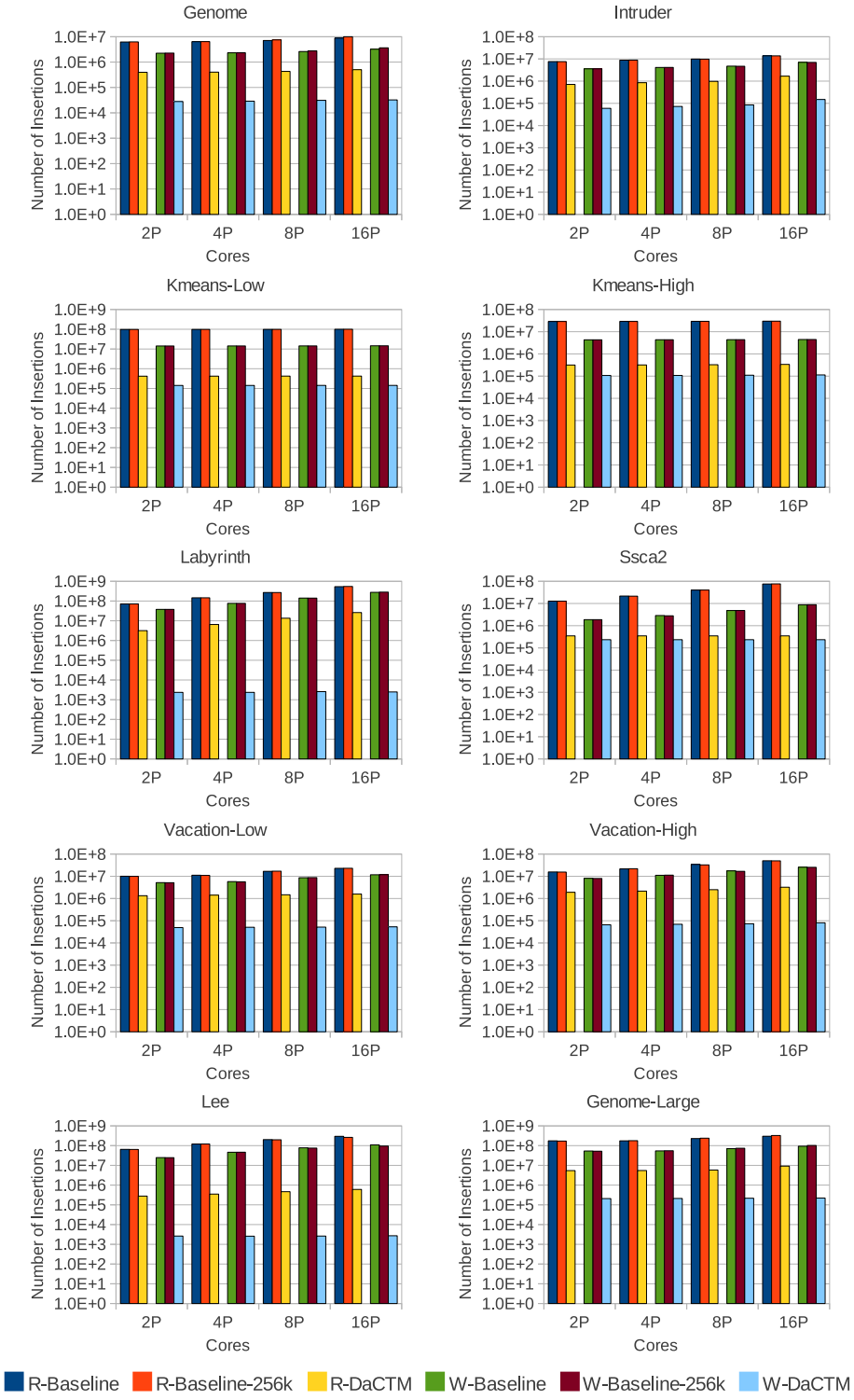

Figure 5.9: Insertions to read/write signatures in the U version of baseline and DaCTM

and Lee (0.20%-0.43%) report the lowest percentage of insertions made to the read signature, in comparison to those made in the baseline. Insertions made to the write signature in DaCTM-U for Lee and Labyrinth applications is negligible, as in DaCTM-CS. A comparably higher percentages of insertions to the read signature, are reported in Vacation-Low (13%, 2P), Vacation-High (12%, 2P) and Intruder (12%, 16P). In the case of write signatures, Ssca2 reports the maximum percentage of insertion which is 12%. Insertions made to the rest of the applications are quite low and in cases like Labyrinth, Vacation-Low, Vacation-High, Lee and Genome-Large it falls to less than 1%, that of the baseline.

It can be observed in both Figures 5.8 and 5.9 in most applications, that the number of insertions made by the baseline architectures increase as the number of cores increase in the system. However this increase is not significant for DaCTM architectures. The reason for this behaviour is that DaCTM produces less false aborts, due to a lower number of insertions made to the signatures. Aborting and restating a transaction, increases the contention of the system. When a system has higher number of cores, the contention for shared resources increases. When the system produces false aborts, this contention gets even higher. When the contention is higher, the idle time of the processor increases, thereby making transactions more susceptible to abort. When a transaction is aborted, its signature is discarded. Each aborted transaction has to be restarted. When it is restarted, it has to create a new signature, because the abort operation has cleared the previous one. This increases the number of insertions made to each signature in the baseline for higher processor counts.

5.5.6 False Positives

As described earlier, signatures produce false positives and inserting more addresses increases the probability of it. The number of false positives produced in DaCTM-CS and DaCTM-U are shown in Figure 5.10 and 5.11 respectively. Each figure also shows the number of false positives produced in the corresponding baseline architectures as well. False positives presented in the figures are measured at the granularity of cache lines. It can be seen in both figures that DaCTM architectures produce either zero or significantly lower number of false positives than those produced in the corresponding baseline architectures.

DaCTM-CS did not produce any false positives in Kmeans-Low, Kmeans-High, Labyrinth, Ssca2 and Lee for all the processor configurations. Both baseline configurations of CS, produced zero false positives for Kmeans Low and High for 2, 4, 8

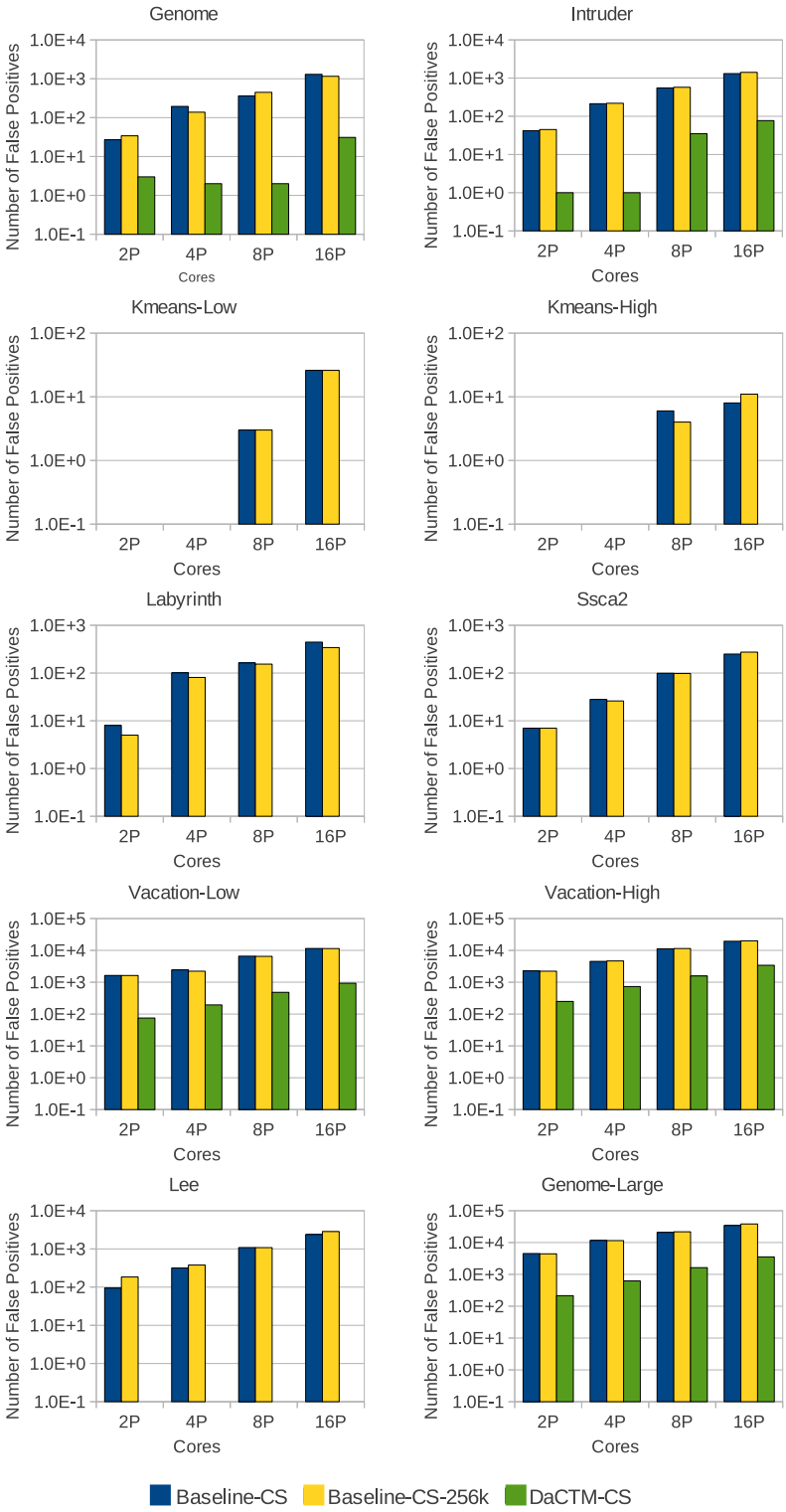

Figure 5.10: Number of false positives presented in the CS version of DaCTM and baselines

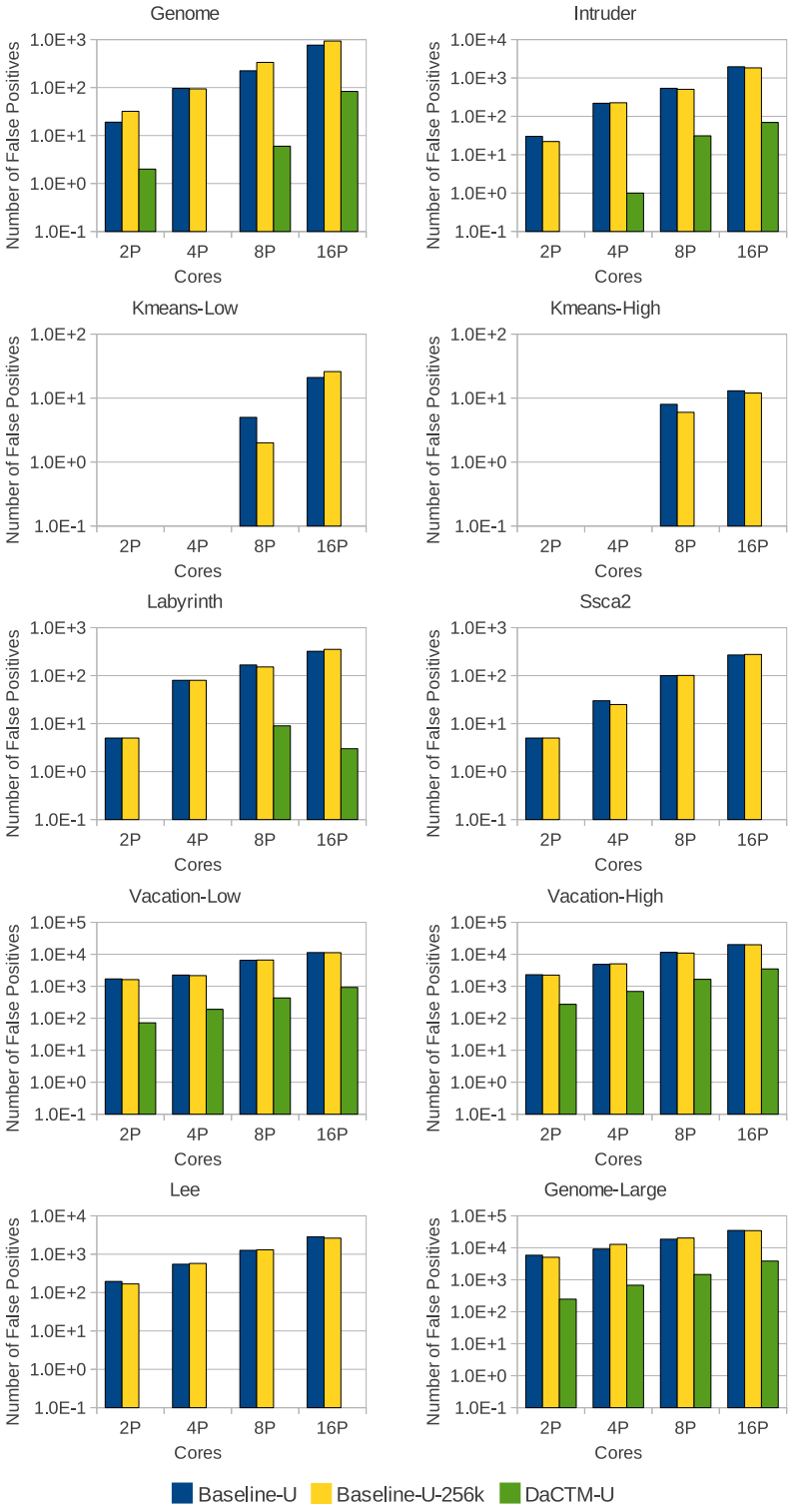

Figure 5.11: Number of false positives presented in the U version of DaCTM and baselines

and 2, 4 processor counts respectively. Among the applications for which DaCTM-CS has produced false positives, the highest percentage in comparison to those produced in the baseline, is reported in Vacation-High which is 18%. The highest percentages reported in other applications are Genome (11%), Intruder (6%), Vacation-Low (8%) and Genome-Large (10%).

In the case of DaCTM-U, Kmeans-Low, Kmeans-High, Ssca2 and Lee did not produce any false positives for all processor configurations. The U version of baseline architecture produced zero false positives only with the Kmeans application (both Low and High) for 2 and 4 processor configurations. Considering only the applications for which DaCTM-U produced false positives, the highest percentage of false positives in comparison to those produced in the baseline is reported in Vacation-High which is 17%. The highest percentages of false positives reported in other applications are Genome (11%), Intruder (6%), Labyrinth (5%), Vacation-Low (8%) and Genome-Large (11%).

Another observation that can be seen in both Figures 5.10 and 5.11, is that the number of false positives produced in the system increases as the number of cores increases. One reason for this is, as the number of cores increases the number of signatures (to check for conflicts) increases as well. More live signatures present in the system, increase the candidates for checking conflicts. Therefore total number of false positives increased as the number of cores increases. However the number of false positives produced per core in both DaCTM architectures remains similar for all the processor configurations for all the applications. The situation is not same for baseline architectures with Genome, Intruder, Labyrinth, Ssca2 and Lee. Among those Intruder, Labyrinth and Lee are applications with high contention. Extra transactions have been added to Ssca2 and Genome to maintain coherence when accessing WNRL data. This makes Genome behave differently than Vacation which has similar characteristics in its original version [74]. All the above mentioned applications can now be considered as having high contention despite their original characteristics presented in [74]. Contention for shared resources increases as the number of cores increases. For an application with higher contention, this could affect it unfavourably. So when false positives are occurring, contention gets further increased and in turn produces more false positives. This does not apply to DaCTM, because they do not produce false positives as in the baseline architectures. Therefore the number of false positives produced per core remains steady.

5.6 Summary

Evaluation of two DaCTM systems and their corresponding baseline systems are presented in this chapter. It described the benchmarks used and the modifications made to to them in order to work with DaCTM and baseline architectures. A description of the simulator along with extensions used to support, Transactional Memory and Memory Regions were also given in the chapter. Performance of DaCTM over baseline architecture and also the scalability of DaCTM was presented thereafter. Finally the chapter characterises the results on various parameters such as processor idle time, bus contention, bus usage, commit phase bus usage, signature insertions and false positives.

Chapter 6

Related Work on DaCTM

This chapter describes some of the work related to the DaCTM proposal. The centrepiece of DaCTM is the hardware support for transactional memory. Also it is the centrepiece of the work presented in Part II and III of the thesis. Therefore the related work in the area of Transactional Memory is presented in Chapter 2. The DaCTM architecture is based on the "data centric" approach to computing. Section 6.1 describes the related work in the area of data centric computing and also architectures that support it. The DaCTM architecture maintains SCC at bulk level and optimises the required hardware operation based on the type of memory location. Sections 6.2 and 6.3 briefly describe related work in cache coherence and memory consistency. Two sections also describe existing work that proposes to distinguish hardware operations based on the type of a memory location. Existing proposals to optimise transactional memory using the access patterns of memory locations are presented in Section 6.4. The chapter also presents, in Section 6.5, several approaches from the memory management literature that uses access pattern information in different situations.

6.1 Data Centric Synchronization

Vaziri *et al.* [106] argue that operation centric synchronization is error prone because it requires programmers to have a clear understanding of which data structures are accessed concurrently, hence propose Data Centric Synchronization (DCS). In DCS a programmer associates synchronization constraints with data structures and the compiler automatically infers points in the program order to preserve consistency. They also present an inter-procedural analysis to determine locations to perform synchronization.

```
Class Customer{
    String city;
    int postCode
    Date date;
    Item item;

    void updateAddress(String c, int z){
        atomic { city=c; postCode=z; }
    }
    void newPurchase(Date d, Item i){
        atomic { date=d; item=i; }
    }
    void newStoreGift(Date d, Item i){
        atomic { date=d; item=i; }
    }
}
```

(a)

```
Class Customer{
    atomic(address) String city;
    atomic(address) int postCode
    atomic(purchase) Date date;
    atomic(purchase) Item item;

    void updateAddress(String c, int z){
        city=c; postCode=z;
    }
    void newPurchase(Date d, Item i){
        date=d; item=i;
    }
    void newStoreGift(Date d, Item i){
        date=d; item=i;
    }
}
```

(b)

Figure 6.1: An example of using data centric approach for synchronization (taken from [106])

For example consider the code segment shown in Figure 6.1(a). There, the class `Customer` has four variables and three methods and all these three methods can get executed in parallel. Therefore the keyword *atomic* has been placed inside all the methods in order to avoid data races. This is the conventional operation centric way of maintaining synchronization. The code segment shown in Figure 6.1(b) shows the "data centric" approach proposed by Vaziri *et al.* [106]. In their approach authors propose to associate the *atomic* keyword with data, instead of associating it with the operations inside a method. The authors also proposed to group data into *atomic sets*, so that atomic constraints can be associated with sets of data instead of associating them with the entire data set used in a program. It can be seen in Figure 6.1(b), that variables `city` and `postCode` are associated with the *atomic set* `address`; variables `date` and `item` are associated with *atomic set* `purchase`. In the proposed framework the complier automatically infers where the locks need to be obtained in order to maintain synchronization. For example, the `updateAddress` method is required only to obtain the lock related to the `address` *atomic set* and the `newPurchase` method is only required to obtain the lock related to the `purchase` *atomic set*. However the `newStoreGift` method is required to obtain locks related to both atomic sets as it accesses data from both of them.

Colorama [13] is an architectural solution to support DCS. The idea is to group data structures into consistency domains and to assign a color to each domain. If a thread accesses a data structure from a domain that it does not own, a critical section is started automatically for that particular thread for that particular domain. No other

thread can now access this domain. In order to realise this proposal at the hardware level, it is required to identify when to start a critical section and when to finish it. In order to achieve the first objective two types of structures are used. One is a shared structure called *Palette* which stores start and end addresses of all the color regions. The second is local to each thread and it stores *colorIDs* of all the regions that the thread owns. A critical section is started when a thread accesses a color region that it does not own. In Colorama, the authors also consider a method as a unit of work as in DCS [106]. Therefore when the execution of a method is completed, the critical section started for that method needs to be finished. They use Transactional Memory as the underlying synchronization mechanism. Later, the same authors proposed Data Coloring [16], a programming model based on the data colouring concept. In this model, data structures are grouped into consistency domains and places are marked (with the *color step* construct) in the program order where data should be consistent.

The concept of associating a *type* with memory locations and inferring required operations to maintain SCC in DaCTM is similar to DCS and Colorama. However both DCS and Colorama only focus on maintaining synchronization, whereas the focus of DaCTM is to maintain SCC as a whole. The memory regions and their assigned types are similar to color regions and color IDs in Colorama. However the *type* for a memory location in DaCTM is derived from the access pattern of that location. In the case of Colorama, the color ID does not represent such information.

6.2 Cache Coherence

Caches have been introduced and placed physically closer to processors to store frequently used data, in order to reduce latency. Memory accesses in a program show temporal and spatial locality. Therefore by storing the most frequently accessed memory locations and their nearby locations, access latency of a processor can be reduced. However, a multi-core chip has more than one processor and each processor has at least one private cache. Due to the presence of locality in memory accesses, more than one processor can access the same or nearby by memory location. When the modifications to these locations are made on local caches of the processors, an erroneous output can be produced unless a mechanism is used to avoid other processors using stale data. The issue that needs addressing here is how to maintain a coherent view of the shared memory.

In order to maintain a coherent view, a coherent detection and enforcement strategy

needs to be implemented. A mechanism to detect coherence violations can be implemented by snooping the bus for read and write requests issued by other processors and comparing them with the state of the local cache entries. As the name implies the state of a cache line indicates the state of it (shared, dirty and so on). In order to enforce coherence, a cache controller can either invalidate the cache entry or update it based on the information gathered by snooping the bus. The coherence protocols which are based on this fundamental concept are called *snoop* based protocols.

In a chip-multiprocessor as the number cores increased, a bus based interconnect cannot support the overall network traffic. Therefore an scalable interconnect like a two-dimensional mesh or an omega network is required. However, snooping is not straightforward in such an interconnect as there can be multiple communication channels among cores. Therefore, those systems rely on a structure called a directory to maintain cache coherence. There, a directory maintains the state about the cache lines accessed by each processor. Read and write operations from all the processors go through the directory. Therefore the directory knows which cache lines are modified and who has the modified cache entry and which cache lines are shared by which processors and so on. Therefore the coherence detection strategy can be integrated with the directory itself. In order to enforce the coherence, when a violation is detected, the directory sends messages to caches which have cached the entry being considered, asking them to invalidate their entries. Since a directory is involved in these sort of mechanisms, they are called *directory* based protocols.

The above paragraphs provides a brief summary of cache coherence protocols. The literature on the subject is quite large and summarising them all here is beyond the scope of the thesis, but interested readers are directed to [68, 101, 103]. However, the rest of this section describes proposals from cache coherence literature that attempt to categorise data and perform coherence selectively on data that requires it.

Ekman *et al.* [31] propose to attach a unit called a Page Sharing Table (PST) in each processor in order to keep track of the pages accessed by the processor and the sharers of those pages. The sharers are stored in a sharing vector, which is broadcast on a separate bus called a *sharing vector bus*. The information in the sharing vector of one processor is used by other processors in deciding which addresses require a tag lookup and which do not. The objective is to reduce the energy consumed, in looking up addresses that are not shared by other processors, in a snooping coherence protocol.

Cantin *et al.* [11] propose *Coarse-Grain Coherence Tracking* which allows processors to send L1 cache misses directly to the main memory without checking those

misses in other processors' caches. Each processor is equipped with a structure called a *Region Coherence Array* (RCA) for monitoring coherence at a granularity bigger than a cache line. This RCA maintains the coherence state of a large aligned memory region. The size of a region is equal to two to the power of number of cache lines. In their approach, a conventional cache coherence protocol is modified so that when a cache is snooped for an address, the RCA of that processor is also snooped. The requesting processor store the response of RCA snooping of the other processor in its RCA. When a processor encountered a cache miss, if checks its RCA to check whether any other processor is caching data in the region that this address falls in to. If that is not the case, the request is directed to memory. The authors argument is that in a conventional coherence protocol, all the cache misses are snooped regardless of whether they are cached or not in other processors. In their approach, snooping is done only for locations that belongs to regions that are cached by other processors. This way their proposal was able to reduce the request latency of a multi-core processor.

Similar to *Coarse-Grain Coherence Tracking*, Moshovos [78] proposed *RegionScout*, a filtering mechanism that dynamically detects non-shared regions. The objective is to reduce energy, latency and bandwidth utilization by avoiding unnecessary snooping and tag lookups. The proposal is to attach two structures, namely a *Not Shared Region Table* (NSRT) and a *Cached Region Hash* (CRH), to each node. A region is defined as an aligned continuous block of memory with size of power of two. As the name suggests NSRT records non shared regions. CRH which is a bloom filter, imprecisely records locally cached regions. When a node needs to issue a memory request, it first check it in the NSRT. If an entry is found, it knows that no other node is caching this, thus broadcasting can be avoided. When a node receives a broadcast request for an entry that is cached in its NSRT, the entry is invalidated. When responding to a memory request, a node uses its CRH to check whether it is caching any entry in the region that the requesting address falls in to. This response is used by the requesting node in deciding whether to insert the entry to NSRT or not.

Zebchuk *et al.* [112] argue that both *Coarse-Grain Coherence Tracking* and *RegionScout* require extra on-chip area which is a scarce resource, therefore hardware designers are unlikely to integrate those [11, 31, 78] into future designs, unless the area and power consumption of those techniques are addressed. As their solution the authors present *RegionTracker* (RT), a framework for coarse-grain tracking without compromising the area or performance of a conventional cache. The RT design proposes to replace the tag array with a structure to facilitate region level lookups. With a single

lookup in RT, it can be determined which and where, blocks of a region are cached. The authors showed that their technique can be used to support the *RegionScout* [78] technique, in order to eliminate unnecessary broadcasts in a snoop coherence protocol, without any resource overhead.

A distributed cached design called R-NUCA which takes the advantage of access pattern information of memory locations is presented by Hardavellas *et al.* [40]. In their approach, accesses are categorised as *Instructions, Data-Shared* and *Data-Private*. Classification is done at the OS level with page level granularity. The advantage is that different block replacement policies can be applied to each category. R-NUCA places private data (*Data-Private*) in the L2 cache slice of the corresponding processor. Once loaded to the memory, *Instructions* remain as read-only. However they are read by many processors. Therefore instructions are replicated in local caches. R-NUCA places shared data (*Data-Shared*) closer to the cores that access them. It also ensures that they are evenly distributed across all tiles and for each shared block there is a unique slice. This way R-NUCA avoids replication, thereby eliminating coherence as well.

Rather than categorising data as shared or private at page-level granularity, *Subspace Snooping* [56] proposes to identify sharers for pages. The argument is that most of the pages are partially shared, *i.e.* shared by more than one core but less than the total number of cores in the system. Therefore categorising a partially shared page as "shared" as in the previously described approaches incurs unnecessary broadcasts and tag lookups in a snoop based coherence protocol. The proposal is to extend a page table entry and the TLB with a sharing vector to record the sharers of each page. For each coherence operation, the requesting processor (*i.e.* the one performing the operation) can find out the sharers of the page to which this address belongs, hence the request is forwarded only to those. Their proposal also provides a feature called *Subspace Shrinking* which allows the removal of obsolete sharers (who does not share the page any more) from the sharing vector. Using this approach the authors were able to reduce unnecessary snooping in snoop based coherence protocols.

Cuesta *et al.*[27] also made a proposal to deactivate the coherence mechanism for private data. Similar to R-NUCA [40] and *Subspace Snooping* [56], in their proposal, the authors also take the advantage of functions available in the OS to distinguish data as private and shared. Initially all the pages are treated as private, hence coherence is deactivated for them. When a processor issues a memory request, which results in a TLB miss, while serving the TLB miss the OS checks whether any other processor

is keeping this page in a private state. If that is the case, the OS issues a coherence recovery notification to the hardware and the coherence is activated for that page from that point onwards. In order to realise the proposal: TLB entries are extended with extra *private* and *locked* bits; page table entries are extended with *private*, *cached* and *keeper* bits.

Literature described in this section presented several attempts from cache coherence literature that aimed to distinguish memory accesses into *shared* and *private* and eliminate the coherence mechanism for the *private* data. One of the objectives of DaCTM is similar to this, that is to reduce the broadcasting and interconnect usage for *private* data. However, in DaCTM *private* data is stored in on-chip memories where as in all the above described approaches they are stored in the globally shared memory. In addition, DaCTM aims to provide Synchronization, Coherence and Consistency as a whole whereas all these proposals only target the provision of coherence.

6.3 Memory Consistency

Integrating memory consistency with cache coherence and providing it at bulk level is one of the objectives of Part I of this thesis. Memory consistency models define the event ordering on parallel processors. They range from strict but easy to understand and reason about sequential consistency (SC) [62] to relaxed but more complex release consistency (RC) [35].

Lamport [62] defined Sequential Consistency (SC) as "*if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program*". This allows programmers to view a program as a collection of operations issued by different processors in the system and these processors are connected to the global memory one at a time by a switch. Since only one processor is connected to the global memory at any given time, the operation performed by it is made visible to others in the system, before the next processor issues its memory operation. This ensures that the order in which these operations are performed, is in accordance with the order specified by the program. In order to provide this simple abstraction, SC enforces constrains on operations performed by each processor. Therefore it cannot take the advantage of certain hardware optimisations such as reordering, buffering, bypassing and so on. Later, several memory models like Release Consistency (RC) [35] which allows relaxations from the constraints imposed by

SC have been proposed. Summarising all of them is beyond the scope of this thesis, but interested readers are directed to the tutorial [2] by Adve and Gharachorloo.

Gharachorloo *et al.* proposed that prefetching and speculation can be used to improve the performance of any consistency model [34]. For prefetching, authors suggest the use of *hardware-controlled non-binding prefetch*. With this approach, data is brought to the cache and kept coherent using coherence protocols. For speculation, a mechanism is required to detect whether speculatively accessed data is correct at the time of using it and also a correction mechanism is required to undo and repeat the computation in case of misspeculation. Hill argues [51] that future multi-core systems should implement SC as the memory model because the support for speculation can narrow the performance gap between SC and RC. He also argues that the performance gained by the relaxed consistency models does not justify the complexity imposed by them. This conjecture of Hill's is validated by the work of Gniady *et al.* [36], in which the authors show that SC can perform as well as RC provided that the hardware has enough support for speculation.

Ceze *et al.* [14] presented BulkSC, an architecture to provide sequential consistency (SC) at block level. The idea is to dynamically group consecutive instructions and to execute them speculatively. Conflicts are checked at the end of blocks and hardware enforces SC at coarse grain level rather than at instruction level. They also achieved performance comparable to a RC implementation. Both of these [14, 36] support Hill's conjecture [51] that a SC implementation can perform as well as RC if enough hardware support for speculation is provided. BulkSC also proposes to operate non-speculatively on private data.

Memory consistency provided by DaCTM is somewhat similar to that of BulkSC. The differences are, blocks are dynamically created in BulkSC whereas in DaCTM they are created statically. Both systems propose to operate non-speculatively on private data. In BulkSC, this private data is stored in the shared memory whilst they are stored in on-chip memory in DaCTM. BulkSC considers all non-private data as shared and hardware is made to maintain consistency constraints on them. In DaCTM all non-private data is further categorised as *Read-Only*, *Write-Now-Read-Later* and *Concurrently-Read-Write* and different consistency constraints are applied on them. The consistency constraint imposed on *Concurrently-Read-Write* data by DaCTM is similar to that of the BulkSC on non-private data.

Dag-consistency is proposed by Blumofe *et al.* [8], which is a relaxation of event ordering based on the *Directed Acyclic Graph* (DAG) of a computation. There, the

authors describe Dag-consistency as follows.

The shared memory M of a multithreaded computation G = (V,E) is dag-consistent if following two conditions hold:

1. Whenever any thread i ∈ V reads any object m ∈ M, it receives a value v modified by some thread j ∈ V such that j writes v to m and there exists no path from thread i to thread j.

2. For any three threads i, j, k ∈ V and there exists a path from thread i to thread j and a path from thread j to thread k, if j writes some object m ∈ M and k reads m, then the value received by k should lastly be written by j not i.

Dag-consistency uses three operations namely *fetch, reconcile, flush* to ensure correct operation. *Fetch* copies a new object from main memory to cache, *reconcile* copies a dirty object from cache to main memory and the cache is flushed using a *flush* operation. The operation on WNRL objects in DaCTM is similar to Dag-consistency. Within a WNRL-transaction, objects are fetched (*fetch*) from the next level memory. When the `TM_END` instruction is executed, all the speculatively modified WNRL objects are written back to the next level memory (*reconcile*) and those cache entries are flushed (*flush*).

6.4 Data Separation in Transactional Memory

Chapter 2 provides literature on the area of Transactional Memory (TM). This section summarises TM proposals which are similar to DaCTM, mainly in the context of categorising data. In Transactional Memory Coherence and Consistency (TCC) [39] the authors propose to use transactions as the unit for maintaining coherence, consistency and synchronization. TCC also proposed to exclude the stack variables from the commit packet, but the work was not extended to identify other local variables using the keywords or access patterns as done in DaCTM. They achieved this by marking certain loads and stores as “local”. Even though this is a straightforward way of categorising data, it limits the code reusability (similar to the naive DaCTM design, see Chapter 4, Section 4.1). DaCTM still uses conventional loads and stores and it is capable of dynamically categorising data (see Chapter 3, Section 3.4, Figure 3.18). Further, all the local variables reside in on-chip memory in DaCTM, thereby having a zero effect on the interconnect. The baseline implementation used for evaluating DaCTM is an

improved version of TCC.

Matveev *et al.* [72] proposed *Virtual Memory Filter* (VMF), a hardware solution that allows Software TM (STM) code to be executed without being instrumented. This is because in STM, transactional code needs to be instrumented to maintain versioning at the software level. With VMF, STM programmers are required only to declare which locations are shared and which locations are unshared. No instrumentation of the transactional code is required. When a transactional access happens to a shared location, VMF detects this and the operation is performed on a shadow copy of the shared location. The objective is to provide fine grain memory tracing for STM, without the effort of instrumenting the code manually. In DaCTM it is also required to define memory regions according to their access patterns similar to deciding shared and unshared locations in VMF. However, the objective of DaCTM is to provide a scalable computing system by coupling a "data centric" approach with TM whereas VMF aims to provide STM code to be executed without being instrumented.

Yen *et al.* [111] present Notary, a signature based hardware TM system. Notary makes two significant contributions. One is a hashing function called *Page-Block-XOR* (PBX) which provides performance similar to H3 [12] hashing but at a lesser hardware cost. The second contribution is a privatization interface that allows a programmer to allocate memory from shared and private heaps. It provides `shared_malloc` and `private_malloc` to access both heaps depending upon usage. In Notary, isolation is dropped for all the memory operations on the private heap. By dropping the isolation on private addresses, Notary achieved a reduction in execution time by reducing false conflicts. They also proposed barriers to allow a shared location to be converted to private and vice versa. DaCTM also categorises data as in Notary. However, in DaCTM, shared data is further categorised (RO, WNRL, CRW) and extra memory allocation functions are declared accordingly. Addresses of all the shared memory locations are inserted to signatures in Notary. In DaCTM addresses of only one type of shared locations (CRW) are inserted to the signature. In Notary, private data is allocated from the shared memory and brought to caches when a need arises whilst they are allocated in on-chip SPM in DaCTM.

Riegel *et al.* proposed [91] to partition data according to their access patterns, so that a STM could implement different concurrency control based on the partition it operates on. In their approach partitioning is done automatically at compile-time/runtime and a programmer is only required to mark the transaction boundaries. Partitions are identified by constructing a Data Structure (DS) graph using *Data Structure Analysis*

[63] techniques. A DS graph is created for each function used in the program, which is used by *Poolalloc* [64] to create pools for DS nodes which have been analysed by DSA for all its uses. These pools are then treated as partitions in the underlying STM. The objective is to use different concurrency control algorithms for different partitions. Even though some of the partitions used in this work [91] are similar to those in DaCTM (*Thread-local,Transaction-local*↔*LO, Read-Only*↔*RO*), the fundamental difference is that DaCTM operates at the hardware level whereas [91] operates at software level.

Sanyal *et al.* [96] present a mechanism to separate shared and unshared data in the heap by setting a flag in the virtual memory page. They also provide a `local_malloc` function as in Notary [111] to access private heaps. In addition to this, they also proposed an algorithm to separate stack variables from being included in the read and write sets of a transaction. The authors propose to add a fully associative buffer named *Local-Undo Buffer* to each processor to preserve the original value of certain local variables. This is because, in [96], isolation is dropped for all the local variables and this could lead to an erroneous output in situations where the first transactional operation is a read and an abort happens after a transactional write is performed on the same location. The authors show that only 1% of accesses in the STAMP suite [74] fall in to that category and propose to preserve the old value of such variables in this *Local-Undo Buffer*. In their algorithm, each memory access is checked against the *Stack Pointer* and *Frame Pointer* registers to determine whether a particular address is in the stack, hence isolation can be dropped for that address. This adds a delay to the critical path. In DaCTM the stack is allocated in the on-chip SPM and each SPM is assigned a range of physical addresses. Therefore when a memory request is issued, in DaCTM, it does not need to be checked as in [96]. In addition, DaCTM does not use any hardware structures like *Local-Undo Buffer*, instead the issue is addressed through the programming model (discussed in Chapter 3, Section 3.4).

The Advanced Synchronization Facility (ASF) [1, 21, 24] is an AMD64 hardware extension for implementing TM and lock free data structures. Seven new instructions have been introduced with ASF. The `SPECULATE` instruction starts an atomic region and the `COMMIT` instruction commits the speculatively modified entries. ASF also provides a `LOCK MOV` instruction that moves data between registers and memory as in a regular `MOV` instruction. However the difference is that a `LOCK MOV` instruction can only be used within an atomic block. ASF hardware performs versioning and conflict detection only for memory locations that are accessed using this `LOCK MOV` instruction. In other

words, within an atomic block, isolation is dropped for memory locations accessed using `MOV` instructions. The objective is to reduce the demand on hardware capacities required to maintain the isolation property. The concept of having separate instructions in ASF is similar to the naive design proposed for DaCTM in Chapter 4, Section 4.1. Such designs cannot be applied in situations where a single function operates on both shared and private data, because during the compilation phase only `MOV` or `LOCK MOV` can be included in the binary.

Memory regions have been proposed by Dean *et al.* [30] as way of achieving strong atomic semantics in STM with performance comparable to a weekly atomic system. Similar to DaCTM, the authors also consider a group of memory locations with the same sharing state as a region. However, the objective of the authors in grouping memory locations into regions is to change the protection status of the entire group with a single, constant-time operation. Using this approach, when a protection state of a region is changed, all the subsequent operations to this region are delayed until all the currently executing transactions on this region finish. DaCTM uses regions so that it can enforce SCC selectively for each region whilst the objective of [30] is completely different to that.

This section summarised the approaches found in TM literature which are mainly focused on categorising data in to different groups. The fundamental difference of DaCTM to those is that, DaCTM further categorises shared data as concurrently shared, non-concurrently shared and read-only shared. Private data was residing in the globally shared memory in all these proposals, whereas in DaCTM those are stored in the on-chip scratch-pad memory.

6.5 Memory Management

This section summarises several approaches, that can be found in memory management literature, which propose to group data of similar access patterns and to allocate them in a suitable memory space. Steensgaard [102] proposed to allocate objects that never escape a thread, in a thread specific heap. Objects that are shared among other threads are allocated in a shared heap. The approach requires an analysis phase to determine which objects are thread-local. The objective of this approach is that, thread specific heaps can be garbage collected separately, thereby reducing the garbage collection latency. Sade *et al.* [93] proposed the use of escape analysis [65, 66, 94] to identify memory allocation requests which are used only by a single thread. They

used that sharing information to extend the Hoard [6] memory allocator with a thread local memory allocation function (`tls_malloc`) to reduce the contention for the global allocator. However these proposals did not focus on communicating this sharing information to hardware in order to improve the cache coherence protocols or memory consistency models as in DaCTM.

The on-chip SPM of a DaCTM processor is a separate physical memory, but it is being mapped to the logical address space of the application. This facilitates a programmer to allocate LO type objects in the on-chip SPM, without the complexity of manually managing separate memories. Similar to this, *Asymmetric Distributed Shared Memory* (ADSM) [33] maps the physical memory of an accelerator in a heterogeneous processor, to the shared logical memory space to allow processors to access objects in the memory of the accelerator. In their approach, when a function is selected for acceleration, all its associated objects are allocated in the accelerator memory which is mapped to the logical memory space. This relieves the programmer from manually transferring data between accelerator memory and the shared memory. In ADSM, only the data objects of functions which are selected for acceleration are allocated in on-chip memory, whilst all the LO type objects are allocated in on-chip SPM in DaCTM.

Part II

SnCTM: Reducing False Transaction Aborts by Adaptively Changing the Source of Conflict Detection

Chapter 7

SnCTM: Adaptive Sources for Conflict Detection

This is the first chapter of Part II of this thesis, which proposes an efficient technique to reduce the number of false transaction aborts in a Hardware Transactional Memory (HTM) system. The chapter describes the concept and the design of SnCTM, an HTM which can adaptively change the source used for detecting conflicts. After showing the motivation for using adaptive sources to detect conflicts with the aid of a preliminary experiment in Section 7.2, the reader is given an overview of related work in hardware signatures in Section 7.3. The concept of SnCTM is described in Section 7.4. Finally, Section 7.5 summarises the chapter.

7.1 Introduction

Commodity chips are now shipped with more than one processor core and in few years time a single chip will include hundreds (if not thousands) of processor cores [52]. It is inevitable that parallel programming becomes the mainstream in order to make use of those cores in the chip. With parallel programming, maintaining mutual exclusive access is one of the issues that programmers have to face. Even though this is achieved via locks in the past, Transactional Memory (TM) [50], a lock free solution based on database transactions [37], has gained attention over the last decade. In TM, during the execution of a critical section, operations are performed speculatively and atomically. All the memory locations that are read/written speculatively, are recorded in a read/write-set respectively. At the end of an atomic block, conflicts are checked using these read and write sets.

Initial hardware TM systems like TCC [39], LogTM [77] propose to keep this read and write set in the Level 1 (L1) cache by extending it with a R (read) and W (write) bit. However this implicitly placed a limitation on the size of a transaction, that is able to fit in the L1 cache. Following the proposal for bulk disambiguation of addresses by Ceze *et al.* [15], Yen *et al.* propose LogTM-SE [110] which suggests the use of hardware signatures to represent the read and write sets of a transaction. A hardware signature is a fixed set of bits, that can be implemented using SRAMs, in which certain bits are set according to the address being considered. The important aspect of using signatures in TM is that, transactions are no longer bounded by the size of the L1 cache. However the disadvantage of using signatures is that they produce false positives. In this context, a false positive refers to a situation where the signature mechanism asserts a conflict, but actually there is not any. False positives lead to false transaction aborts and this degrades the performance of a TM system.

Several proposals [19, 20, 61, 84, 85, 86, 111] have been made to reduce the number of false positives that occur in a hardware TM system. All of these approaches focus on the design and the implementation of signatures in hardware. Part II of this thesis aims to address the issue of reducing false positives from a different angle. As an entry point to the discussion, consider the following question. *Can the usage of signatures, in detecting conflicts, be reduced ?* The reason for raising this question is, if the use of signatures to detect conflicts can be reduced, then the false positives can be reduced. Then the obvious follow-up question would be, *if signatures are not being used, what else can be used to detect conflicts ?* The answer is cache lines. So the proposal of Part II of the thesis is to use both cache lines and signatures to maintain the read and write set of a transaction. When this approach is followed, if the size of a transaction fits in the L1 cache, the cache line information is used to detect conflicts. Signatures are used otherwise. To this end, Part II of this thesis proposes SnCTM: a hardware transactional memory system that adaptively changes the source used for detecting conflicts.

Part II of this thesis makes following contributions.

- The concept of adaptively changing the source of information used to detect conflicts in a hardware TM system, is introduced. It also shows how an existing TM architecture can be extended to support the SnCTM concept.

- The performance evaluation of SnCTM shows improvements of up to 4.62 and 2.93 times speed-up over a baseline TM using lazy versioning and lazy conflict

detection (an improved TCC [39]) with two commonly used signature configurations.

- SnCTM gives the opportunity to reduce the size of a signature without compromising the performance. A sensitivity analysis shows that SnCTM with a 64 bit signature can deliver performance comparable to a perfect signature of 8k bits.

7.2 Motivation

Most of the TM systems (eg: LogTM-SE [110], SigTM [75], VTM [89]) propose to use signatures to record read and write sets of a transaction. This facilitates a transaction to have an unbounded amount of speculative data. Here, the term 'unbounded' means that a transaction is not bounded by the size of its local cache. This is because most of the initial TM systems like TCC [39], LogTM [77] propose to keep the read and write set of a transaction in the Level 1 (L1) cache. Some of the TM systems like LogTM-SE [110] that support unbounded transactions, also support virtualizable transactions, meaning that transactions can even be longer than the scheduling quanta. However the support for virtualizable transactions cannot be provided by only having signatures to record read and write sets of a transaction, thus requires support from the runtime system. Therefore the discussion is only focused on TM systems that use signatures to support an unbounded amount of speculative data.

Early HTM system like TCC and LogTM propose to extend L1 cache with Read and Write (R and W) bits to record the read and write sets of a transaction. This requires transactions to be bounded by the size of the L1 cache of a processor. Bulk [15] proposes to encode this information into a fixed sized hardware 'signature'. This approach allows the size of a transaction not to be bounded by the size of the L1 cache. Signatures have the disadvantage of producing false positives. That is, when tested for the membership of an address in a signature, it may assert positive even if the address is not present in the signature. False positives lead to false transaction aborts, thereby degrading the performance of a transactional memory system.

False transaction aborts which are caused by false positives can be reduced by optimising the implementation of a signature. A number of approaches to achieve this are discussed in Section 7.3. Part II of this thesis takes a completely orthogonal approach to those and proposes a simple hardware solution to reduce false transaction aborts.

In order to make a case for the proposed scheme, the following question is considered initially. "*Does an HTM system require signatures all the time, to detect conflicts ?*". The answer is "*No*". This is because, not all the transactions exceed the size of the L1 cache. When this is the case, the read and write sets can be kept in the L1 cache of the processor. For such transactions, there is no need to use signatures. In order to validate the above answer and to get an intuition of how many transactions actually needed a signature, a preliminary experiment is made with a lazy-lazy HTM system, similar to TCC [39], using 2-16 cores. The experiment was carried out with two signature bit widths (1024, 2048) which are the sizes generally used in hardware TM experiments.

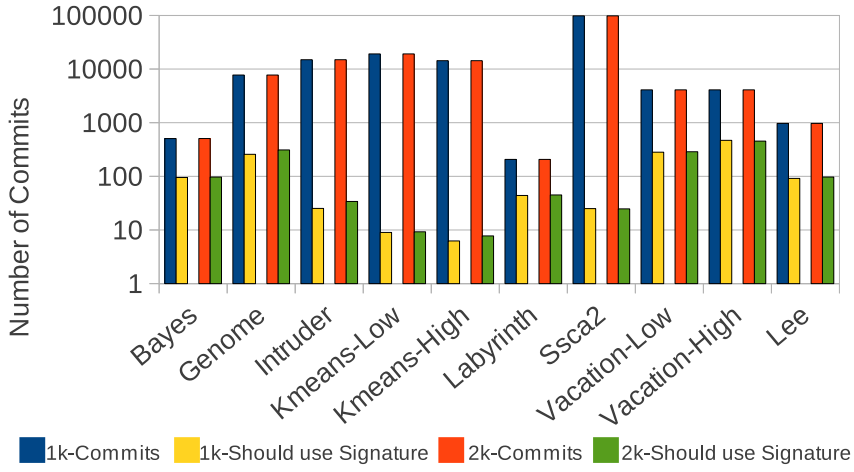

Figure 7.1: Signature requirement for transactions committed

Figure 7.1 shows the number of transactions committed and from those commits how many actually needed a signature. In the legend *1k-Commits* represents the number of commits made in the system with 1024 bit signature and *1k-Should use Signature* represents the number of commits that actually require a signature mechanism to detect conflicts in the same system. When the legend has *2k* instead of *1k*, the same definition applies to a system with 2048 bit signature. The corresponding values are the average of 2-16 cores.

For both signature configurations, it can be seen that the number of commits that require a signature is either low or negligible. The disadvantage of using signatures is that they produce false positives. The same experiment is also used to measure the amount of false positives that could have been avoided if the signatures are not being used for situations where the read and write set fits in the cache. The following mechanism is used for this measurement. The simulator is equipped with a monitor mode

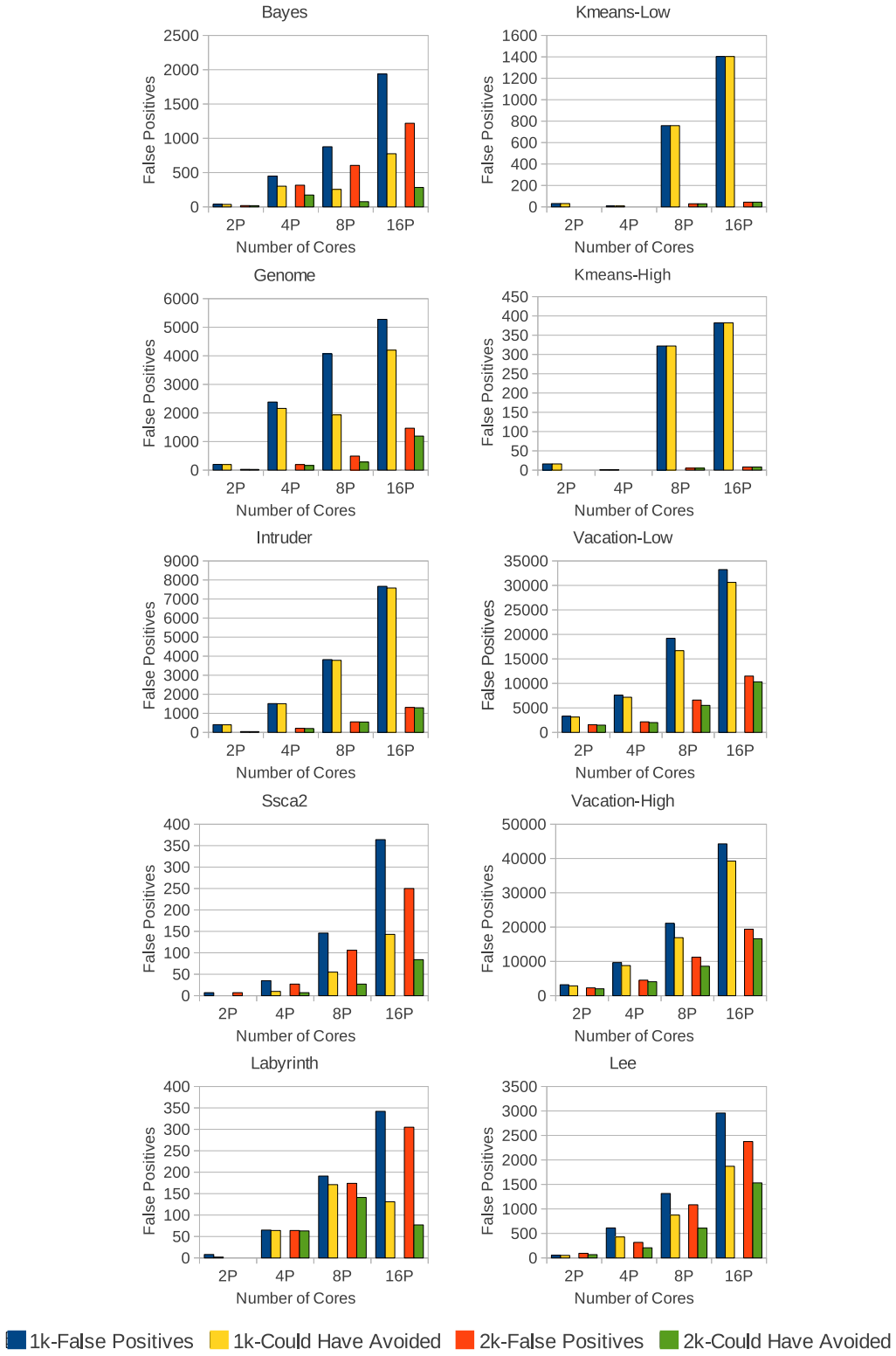

Figure 7.2: False aborts that could have been avoided

which can take certain statistics without affecting the timing model of the simulation. When the conflict detection phase asserts an abort, the monitor mode uses its internal data structures to determine whether the abort is a true abort of a false abort. In the case of a false abort, the monitor mode also checks whether a signature is required for this commit. If a signature is not required, it marks this abort as an abort that could have been avoided.

Figure 7.2 shows the number of false transaction aborts produced in the system and how many of them could have been avoided if the signatures are being used only for situations which require the use of it. Here in the legend *1k-False Positives* represents the number of false positives that occur in a system with 1024 bit signature and *1k-Could Have Avoided* shows the number of false positives that could have been avoided if signatures are not being used. Similarly when the legend has *2k* instead of *1k*, the same definition applies to a system with 2048 bit signature. In the same figure, the X axis represents the number of processors in the system (2P$\rightarrow$2 processors, 4P$\rightarrow$4 processors and so on). It can be seen from Figure 7.2 that majority of false positives could have been avoided if the signatures are only used in cases which requires to do so.

Using this observation from the preliminary experiment as the basis, the hypothesis of Part II of the thesis is formed. That is, the execution time of a TM application can be reduced by reducing false aborts by means of changing the source of the information used to detect conflicts. The term "changing the source of information" means adaptively using signatures or the 'R' and 'W' bits in the cache line, to detect conflicts.

7.3 Related Work on Hardware Signatures

Transactional memory (TM) [50] has been proposed as a way of achieving optimistic concurrency in parallel programming. In Transactional Memory Coherence and Consistency (TCC) [39] the authors propose to use transactions as the unit for maintaining coherence, consistency and synchronization. Since then, many TM approaches have been proposed and Chapter 2 provides a good summary on key HTMs. It also provides a good overview on semantic and performance considerations of TM systems. This section only focuses on literature related to using signatures in TM systems.

Ceze *et al.* [15] defines a hardware signature as a fixed bit width representation of a set of addresses. The objective of Ceze's proposal, *Bulk*, is to produce a hash value by encoding all the access information of a thread, so that caches remain unmodified.

In order to insert an address to a signature, a hash function is performed on the address and a logical OR operation is performed with the hash value and the existing signature. The authors also formally define the operations involved with signatures. The $\cup$ operation, which is similar to a union operation, is used to combine several signatures together. This is useful in supporting nested transactions as the signature of a child transaction can be combined with the parent using this operation. The $\cap$ operation is used to perform an intersection operation among two signatures. This can be used to check if two signatures have at least one item in common. That is by intersecting two signatures and checking if the resulting signature is empty. In order to check whether a signature is empty, the authors define another operation, $= \theta$. In order to check the membership of an address in a signature, the ε operator can be used. This is done by, first generating a temporary signature using the given address, then intersecting it with the real signature, and finally checking if the resulting signature is empty. The last operation, δ, is used to decode a signature into the set of addresses that are being used to generate it. This is done by first generating cache indexes that could set the corresponding bits in the signature, and checking the cache to validate whether those entries have actually been accessed. *Bulk* proposes to use simple bit permutations on an address as its hash function in order to generate a signature.

LogTM-SE [110] also uses signatures to maintain read and write sets of a transaction. Their signature implementation is based on selecting different bits of the address. LogTM-SE comes with three signature implementations. The bit-select (BS) scheme takes the n least significant bits of a block address and produces a signature of size $N = 2^n$ bits. The double-bit-select (DBS) produces a signature by combining two BS implementations. The first one takes the first n bits and the second one takes the second n bits. The resulting signature is of size $2^{(n+1)}$. The third scheme, coarse-bit-select (CBS), produces a signature by taking the n least significant bits of a macro block (block of 1KB is used in their experiments). The authors suggest to use this scheme for large transactions.

SigTM [75] which is a hybrid TM system, also uses hardware support to maintain signatures. They use combination of permutation and bit shifting as their hash functions. As SigTM is a hybrid TM system, it also provides user-level instructions to manage signatures. SigTM supports four hash functions: (1) unpermuted cache line addresses; (2) permuted cache line addresses as in Bulk [15]; (3) shifting the output of step 2 by 10 bits; (4) permuting the 16 least significant bits of a cache line address. The authors also conclude that the choice of the hash function can play a significant

role in producing an accurate signature.

Sanchez *et al.* concluded [95] that the H3 [12] class of hash functions should be used in signatures instead of bit selection as in previously proposed signature implementations [15, 75, 110]. The authors also showed that using k single ported SRAMs (parallel bloom filters) to implement signatures is an area efficient technique. The authors also show that there is a relation between the number of hash functions, number of addresses inserted and the probability of producing false positives. That is, for large transactions, a signature with smaller number of hash functions tend to produce lower false positives. When the transactions are smaller, a signature mechanism having higher number of hash functions tend to produce low false positives. They propose a technique which they call Cuckoo-Bloom signatures, which perform better for both smaller and larger transactions. They use a table which stores three hash values for a given address. When inserting an address, three hash functions are used, resulting three hash values. The first two are used to index this table whilst the third provides extra information about the address, so that the entire representation becomes more accurate. However implementing Cuckoo-Bloom signatures in hardware is complex and it does not support the intersection operation as well.

Yen *et al.* [111] argue that H3 implementations use many XOR gates, thus increasing the area and power overhead of signatures. They propose the Page-Block-XOR (PBX) hash function that delivers performance similar to an H3 hash function, but at a lesser hardware cost. Their proposal is motivated by exploiting the randomness of addresses. In addition to introducing a new hash function, the authors also proposed a filtering mechanism to reduce the number of addresses being inserted to signatures. This is provided via a new privatization interface which provides memory management functions to allocate and deallocate memory from shared and private heaps. Accesses to the private heaps are not added to the signatures and isolation is dropped as well. By reducing the number of insertions made to a signature, Notary [111] was able to reduce false aborts thereby reducing the execution time of an application.

Quislant *et al.* [85] propose to take advantage of locality of memory references to design hardware signatures. The authors observe that false aborts can arise due to address aliasing and filter occupancy. When a transaction is small, it only occupies a small fraction of the signature, but due to address aliasing it can introduce false positives. When a transaction is bigger, the filter occupancy is higher and this leads to false aborts as well. In order to reduce false aborts incurred by the first case, in the proposed scheme, nearby memory addresses only share some bits in the filter. By

exploiting the locality of addresses, their proposal is able to reduce false aborts which arise due to address aliasing. However, the system still has false aborts arising due to filter occupancy.

Choi *et al.* [19] present an interesting fact, that is sometimes false positives in a signature based TM system can be helpful as well. They argue that, when a signature asserts an abort erroneously, it could also be the case that this transaction is meant to be aborted in future due to a real conflict. Therefore if a transaction aborts early because of a false abort, it could save the wasted work that it would have been doing from the false abort to the real abort. They categorise these false aborts as "good" and use this early conflict detection to improve the performance. Their proposal is based on an observation that there is a relation between the good/bad false positives and the granularity of bits used in the address. In order to exploit this good and bad positives, the authors propose *Adaptive Grain Signature* which changes the granularity of the bit range input to the hash function. Their mechanism uses an Abort History Table (AHT) which contains the starting addresses of transactions which have aborted others. The output (hit/miss) of the AHT is fed to a multiplexer which decides the range of bit field to be used in the rest of the addresses accessed within that transaction. Using this approach the authors were able to increase the number of performance friendly false positives and reduce the number of performance destructive ones.

Labrecque *et al.* [61] propose to use reconfigurable signatures. Their approach is to customise the signature to match with the access pattern of the application and to minimize the false conflicts. In order to use their approach, first a trie is constructed using the memory addresses obtained from a trace of an application. Each leaf of the trie represents a bit in the signature. As the number of memory accesses is large, this initial signature will have a higher bit width. Thereafter the trie is truncated by selecting the most frequently accessed branches. After this stage the false positives of the trace are calculated and the trie is expanded with additional branches to reach a desired false positive rate. The signature bits that do not affect the false positive rate, are removed thereafter, resulting in the final signature.

Quislant *et al.* proposed multiset signatures [86], which combine read and write signatures into one. In this manner the size of the signature is doubled without adding any extra hardware. As both read and write signatures are combined, each bloom filter is equipped with two hash functions, one for read operations and one for write operations. This is required to distinguish between read and write operations. However

when it comes to the implementation, this poses a challenge as it now requires 2-ported SRAMs instead of single-ported ones. This doubles the area requirement of a signature. In order to overcome this issue, the authors propose to use *i* double-ported SRAMs and *k-i* single-ported SRAMs to implement the combined signature. In this manner, if *i* is low, the area requirement is similar to the original. Their approach is also enhanced using locality sensitive signatures [85] proposed by the same authors. However the authors did not describe a mechanism to avoid the false aborts that can be caused by read-read dependencies.

Concurrently with the work of Quislant *et al.* [86], Choi *et al.* propose the use of a single signature (*unified-signature*) for both read and write sets [20]. Similar to the multiset signature [86], the *unified-signature* is able to double the size of the signature by combining both read and write signatures. By having a larger signature without any hardware cost, they were able to reduce the false positives, thereby increasing the performance. However the authors did not consider the area cost as Quislant *et al.* [86] did. Unlike the multiset proposal [86], the authors consider the false aborts that can arise from two transactions reading the same memory location. In order to reduce the impact from these read-read dependencies signalling conflicts, they also proposed to use a small helper signature alongside the unified signature. The helper signature works as a write signature, but is smaller in size. When detecting conflicts for *read-exclusive* requests, only the main signature is checked. When detecting conflicts for *read* requests, both the main and the helper signature is checked. For the latter case, a conflict is said to occur if both signatures asserted the membership of the address.

Instead of using fixed size signatures, Orosa *et al.* [84] propose to dynamically assign resources to a signature. The objective of the proposal, *FlexSig*, is to lower the false positive rate by redistributing the available hardware among signatures. *FlexSig* does not have fixed number of signatures. Instead, it comprises *T* bloom filters, with the capability of providing one to *T* signatures. Since signatures are created at runtime, *FlexSig* also comes with an allocation and deallocation algorithm. When a thread requires a signature, the request is forwarded to the allocation routine, which then produce a signature combining one or more bloom filters. If there are no bloom filters available it will free some of the existing ones to accommodate the current request. A signature is composed of one or more registers (64 bits) which has its own hash function. When a signature wants to record an address, each register performs the hash function on the address and a bit is set in their register. In order to check the membership of an address, hash functions of each register are applied to the address

and each register checks whether the corresponding bit is set or not. As the bloom filters are independent of each other, they can be added/removed from a signature without comprising the correctness.

Even though the objective of SnCTM is similar to many of the above, that is to reduce false aborts, it differs from all these approaches. Firstly because it is not another signature implementation. The proposal is to use it only when there is a need to do so. Secondly it is not tied to any signature implementation, therefore any of the above mentioned signature implementations can be used as the underlying signature mechanism in SnCTM.

7.4 SnCTM Concept

The objective of SnCTM is to adaptively change the source of information used during the conflict detection phase in a Hardware Transactional Memory (HTM) system. The concept of SnCTM is described in this section. When signatures are used to detect conflicts in HTM systems, they can produce false positives. However signatures are required for detecting conflicts, only if a particular transaction has speculative data that cannot be stored within its L1 cache.

In the proposed approach, whether to use cache line information or signatures to detect conflicts is decided at the time of committing. This decision depends on the overflow status of currently running transactions. In this approach, when a transaction is going to commit, the committing processor needs to check whether any of the other processors have encountered a cache overflow during the speculative execution. If that is the case, the write signature of the committing processor is communicated to the others and they check their read signatures with the received one to detect conflicts. If none of the concurrently running transactions have encountered a cache overflow, then there is no need to use signatures to detect conflicts. Therefore the committing processor communicates its write-set to the other processors using the ‘W’ bit information in its cache line. When they receive this ‘W’ bit information, other processors check it with their ‘R’ bit information in the cache lines.

The communication of the commit message and the conflict detection phase needs to be generalised to adaptively change between cache lines and signatures, in order to support existing HTM proposals. This can be done by including a flag in the header of the commit message. When a processor is going to commit, it creates the commit message either using the signature or the cache line information and the flag is set

accordingly. When a processor receives this commit message it first reads this flag and determines what source to use to detect conflict. Figure 7.3(a) illustrates concept of SnCTM with respect to a committing processor and Figure 7.3(b) shows it from the receiving processor's view.

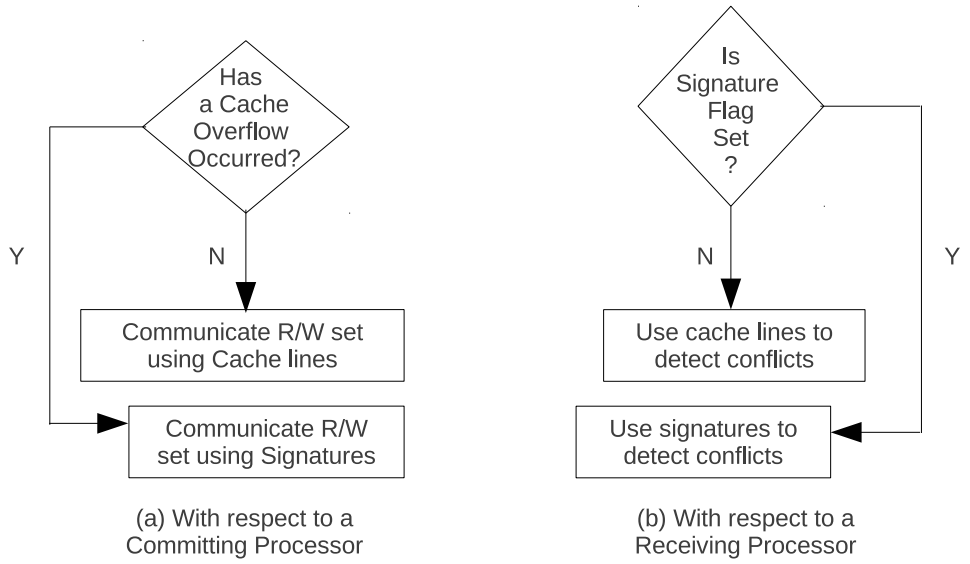

Figure 7.3: The concept of SnCTM

In order to realise the SnCTM concept, a mechanism is needed to maintain the overflow status of the processors in the system. This can be done by having a local flag in each processor and setting it when a transaction overflows. In this approach the committing processor needs to communicate to all the other processors before initiating the commit phase, in order to decide whether to communicate cache line information or signature. A centralized bit map, in which each processor sets the corresponding bit when overflowing, can be used for this as well. In that case the committing processor can check this bit map and decide what source to use to detect conflicts, without communicating to other processors.

Another aspect that needs to be considered when realising the SnCTM concept is, when to update signatures and 'R' and 'W' bits in cache lines. In the case of signatures the term "update" refers to, inserting an address to the signature. In the case of cache lines, it refers to setting 'R' and 'W' bits. The most simple approach is to maintain read/write sets and signatures simultaneously. That is when the 'R' bit is set in the cache line, that address is also inserted to the read signature and the same applies to writes that set the 'W' bit in the cache line. Another approach is to keep the read and write set in the cache line and to compose it to the corresponding signature only if the

committing processor asks to use signatures. Regardless of the method used to maintain signatures and cache line bits, the SnCTM approach guarantees that signatures are being used to detect conflicts only if it is necessary. In this manner SnCTM aims to keep the number of false transaction aborts to a minimum level, thereby reducing the execution time of an application.

7.5 Summary

This chapter presented the concept of SnCTM, a novel way of reducing the false aborts by adaptively changing the source used during the conflict detection stage. The idea is to decide at the time of committing which source to use, *i.e.* cache line or signature. This way the use of signatures is limited to situations where speculative data cannot be held in the local cache. The chapter starts the discussion by presenting a motivating example using a preliminary experiment, which shows the need for having an adaptive mechanism to change between cache line information and signatures to detect conflicts. A comprehensive summary of hardware signatures is provided thereafter, followed by the description of the SnCTM concept. The important aspect of the SnCTM proposal is that it is not tied to any specific signature implementation. Therefore all the signature optimising mechanisms proposed in the literature can be integrated with SnCTM.

Chapter 8

SnCTM Implementation and Evaluation

This chapter describes the architecture of a SnCTM processor and the evaluation of it, in terms of performance. Architectural extensions required to support the SnCTM concept are described in Section 8.1. The performance impact of reducing false transaction aborts using the SnCTM approach is presented in Section 8.2. Finally, Section 8.3 summarises the chapter.

8.1 SnCTM Architecture

This section describes an architecture that supports the SnCTM concept. First it takes an existing HTM as the baseline architecture and later it shows how this baseline can be extended to realise the SnCTM concept.

8.1.1 Baseline Architecture

An improved version of Transactional Memory Coherence and Consistency (TCC) [39] is used as the baseline architecture. The baseline described here is similar to the one described in Chapter 4 (Section 4.5). For the sake of completeness, a brief description is given in the reminder of the section. The transactional memory implementation in this baseline is similar to any other lazy-lazy hardware TM system. When the `TM_BEGIN` instruction is executed, a flag (*IN_TX*) is set. When this flag is set, all the subsequent operations are performed speculatively until the `TM_END` instruction is executed. In order to provide an unbounded amount of transactional data, the baseline

uses hardware signatures [95] to maintain the read and write sets, using parallel bloom filters to increase accuracy. Since the baseline architecture is based on TCC which does not implement any coherence protocols, transactions are used to maintain coherence and consistency as well. Therefore at the end of a transaction, the next level memory copies are updated and local copies which are read/written are flushed. This is necessary because, local caches may end up keeping stale data due to the fact that no conventional coherence protocols are used.

When a processor needs to commit a transaction, it first requests commit permission from a centralised *commit-arbiter*. Commit permission is granted based on a least recently granted policy. Once the commit permission is granted, the committing processor broadcasts its write-signature to all the other processors. Upon receiving this write-signature, each processor performs a bitwise AND operation with their read-signature. If all the hashes in the resulting signature are non-zero, then it is considered as a conflict and the processor aborts. Figure 8.1 shows signature operations used in the baseline architecture. Figure 8.1(a) shows performing an AND operation between two signatures and Figure 8.1(b) shows how to check whether all the resulting hashes are zero.

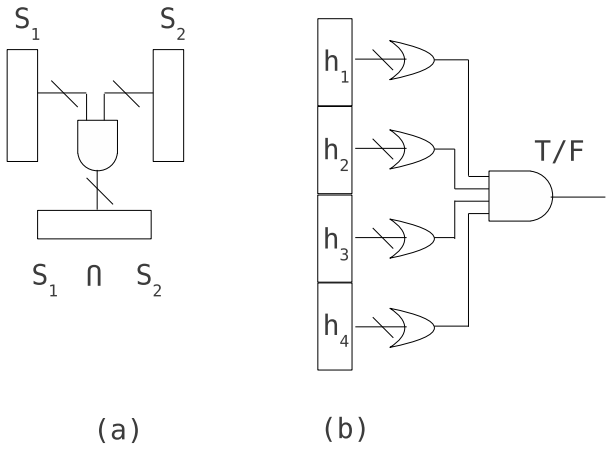

Figure 8.1: Signature operations used in SnCTM

After sending the write-signature to all the other processors, the committing processor updates the next level memory (either Level 2 (L2) cache or main memory) with all the speculatively modified values. During this commit phase, the communication arbiter denies any request to use the interconnect. Once the next level memory is updated with all the speculatively modified cache entries, all these entries need to be flushed and both read and write signatures need to be cleared as well. A more

elaborative description on signatures is presented in Chapter 4, Section 4.3.3.

The baseline used for the evaluation operates under the lazy-lazy TM principle, *i.e.* conflict detection and version management happens lazily. Since version management is done lazily, the speculatively modified data is held in L1 cache and the unmodified data is held either in L2 cache or in main memory. Therefore when an speculatively modified cache entry needs to be evicted for capacity reasons, special measures are required to maintain the isolation property of TM. This is addressed in the baseline by serialising overflows. That is, when a cache entry needs to be rejected while a processor is inside a transaction, permission is sought from the *overflow arbiter*. Overflow permission is also granted based on a least recently granted policy. Once the overflow permission is granted, the processor flushes the cache line from its L1 cache and updates the corresponding entry either in the L2 cache or the main memory. Each processor has a register called *Overflow_Status*, which is set once the overflow permission is granted. Once this flag is set, the processor does not need to seek further permission. A processor needs to ask for overflow permission only if the cache line is modified during the current transaction. An extra 'W' bit is used to mark all the speculatively modified entries. A dirty bit is not sufficient for this purpose because the entry could have been dirty due to a write operation performed outside a transaction. Therefore the baseline architecture cannot eliminate both 'R' and 'W' bits that were present in the original TCC. It needs to keep the 'W' bit to indicate that this cache line has been modified during the transaction. If the 'W' bit is not set, there is no need to seek overflow permission. If an overflow request is denied, the processor stalls until the request is granted.

Even though the *commit-arbiter* of the baseline operates on a *least-recently-granted* policy, there is an exception to this for processors who have transactional cache overflows. That is, once the overflow permission is granted to a processor, all the commit requests from other processors are denied, until the overflowing processor commits. This is because once a speculatively modified entry is written back to the next level memory, either L2 cache or main memory, the old value is lost and this is a non-reversible action. Allowing cache overflows to speculatively modified entries, can be considered as violating the atomicity and isolation properties of the lazy-lazy transactional memory. This is because the overflowed cache entries can now be read by other processors before the current transaction commits. In order to maintain the consistency of the system, the current transaction of the overflowing processor, is made unabortable. This is achieved, as described earlier, by denying all the commit requests

until the overflowing processor commits. This policy works correct because the protocol simply follows a *committer-wins* policy, in which the committing transaction progresses in case of a conflict.

8.1.2 SnCTM Design

This section describes how to extend the baseline architecture, described in Section 8.1.1, to realise the SnCTM concept. As described in Chapter 7 (Section 7.4), the basic idea of SnCTM is to adaptively change the source of information used to detect conflicts during the commit phase. Since a processor does not know in advance which source can be used to detect conflicts, hardware should have the capability to store read and write sets in both formats, *i.e.* 'R' and 'W' bits in cache lines and signatures. Since SnCTM does not use any cache coherence protocol, by reusing the the existing entry for the state field of the coherence protocol to keep 'R' and 'W' bits, area utilization of L1 cache can be kept unchanged. A complete SnCTM system is shown in Figure 8.2. The only addition to the baseline, apart from the control logic, is the 'R' bit field in the cache lines.

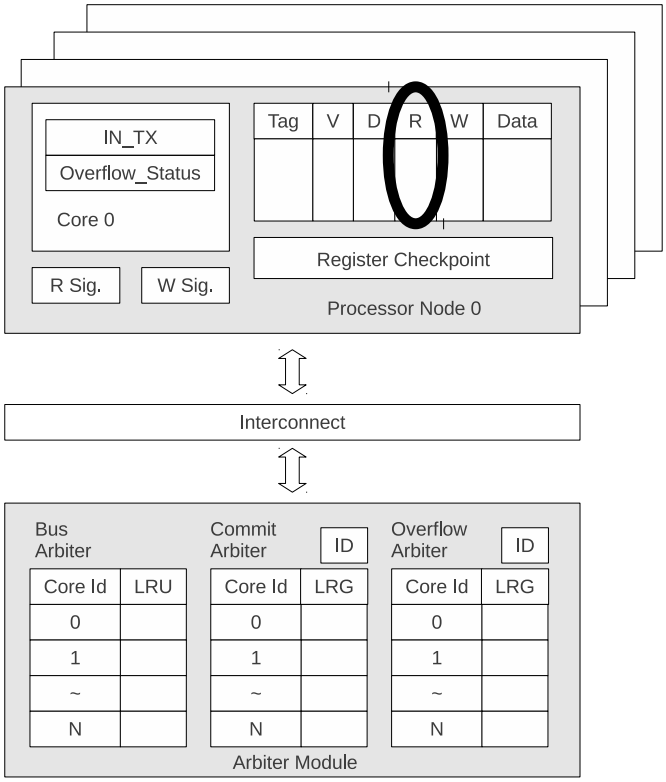

Figure 8.2: A complete SnCTM system

When a processor is executing a transaction, all the read operations set the 'R' bit in the cache line and also they set the corresponding bits in the read signature. The same applies to the write operations. In this way each processor keeps both the sources updated and one of them is used during the commit phase. Due to the nature of the baseline used, at any given time only one processor can be granted overflow permission. Also the commit protocol of the baseline prevents any other processors committing before the overflowing processor. When the commit permission is granted, without any communication to other processors, the committing processor itself can decide whether to use signature or cache lines. This is because if the committing processor is not the overflowing processor, there cannot be any other processor in the system which has been granted overflow status. If there is any other processor which has been granted overflow permission, then this processor cannot be granted the commit permission.

Therefore, if the committing processor has been granted overflow status, it broadcasts its write signature to other processors and they check it with their read signatures to detect conflicts. If the committing processor has not been granted overflow status, this means the transaction was able to fit in the L1 cache. Therefore it can use the 'R' and 'W' bit information in the cache line to detect conflicts. In this case, the committing processor broadcasts its write set to other processors and they check it with their read set to detect conflicts. In order to adaptively decide which source to use in the conflict detection mechanism of the receiving processor, the commit message includes an extra flag called *Type* which notifies the receiving processor about the type of information it carries, *i.e.* signature or cache line.

In order to set the *Type* flag of the commit message, the (*Overflow_Status*) flag of the baseline is used. If set, the signature is included in the commit message and the *Type* flag in the message is set. If unset, the cache line bits are used and the *Type* flag of the commit message is kept unset as well. Once the other processors receive this commit message, this *Type* flag is checked and the corresponding source for detecting conflicts is determined. Figure 8.3 shows the mechanism used in a SnCTM processor when checking conflicts as a response to a commit message.

8.2 Evaluation

The evaluation of SnCTM is presented in this section. After discussing the evaluation setup in Section 8.2.1, the performance evaluation of SnCTM is presented in Section

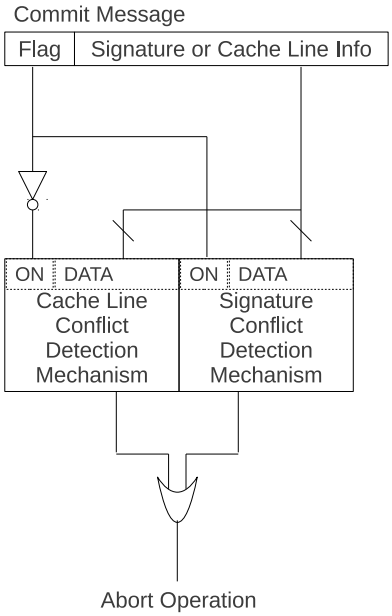

Figure 8.3: Adaptively checking for conflicts in a SnCTM processor

8.2.2. The same section shows that a SnCTM system can outperform an improved version of TCC [39], with two commonly used signature configurations. Characterization of the results of SnCTM is presented in Section 8.2.3. The sensitivity of SnCTM and the baseline to different signature lengths is shown in Section 8.2.4.

8.2.1 Evaluation Setup

The simulation environment used for evaluating SnCTM is similar to the one used for evaluating DaCTM, described in Chapter 5, Section 5.3. For the sake of completeness, a brief description of the system is presented in the remainder of the section, readers are directed to Chapter 5 for a more elaborative description.

In order to evaluate the SnCTM architecture, a lazy-lazy hardware transactional memory system is modelled in Simics [70], a full system simulator running Linux kernel version 2.6.16. The SnCTM system is configured with the components shown in Table 8.1.

Lee's routing algorithm [108] and applications from the STAMP [74] benchmark suite were used to evaluate the SnCTM architecture. For comparison purposes, all the applications were also executed on the baseline architecture. However, due to the fact that no cache coherence protocol is implemented in the baseline or SnCTM, most

Component	Feature
Processors	1-16, in-order
L1 Data Cache	2 way assoc, 64 B line, 32 KB size, 2 cycle latency, private per core
Signature	1024, 2048 Bits, 4 Parallel H3 [12] Hash functions
L2 Data Cache	8 way assoc, 64 B line, 4 MB size, 20 cycle latency, shared
Interconnect	Split-transaction bus, 4 cycle latency, 64 B data width
Main Memory	100 cycle latency

Table 8.1: Components and features of the SnCTM evaluation environment

of these applications were not able to execute without being modified. This is because, these applications access shared data outside transactions. As the baseline and SnCTM provide coherence using transactions, some of the applications were modified by adding extra transactions in places where they access shared data. No modification was required for Vacation, Labyrinth and Lee as they do not access shared data outside transactions. Smaller transactions similar to the already existing ones have been added to Intruder, Genome and Kmeans. The only significant change has been made to Ssca2 by adding several large transactions as the majority of the non-transactional code accesses shared data.

Application	Input
Genome	-g256 -s16 -n16384
Intruder	-a10 -l4 -n2038 -s1
Kmeans-Low	-m40 -n40 -t0.05 -i random-n2048-d16-c16.txt
Kmeans-High	-m15 -n15 -t0.05 -i random-n2048-d16-c16.txt
Labyrinth	-i random-x32-y32-z3-n96.txt
Ssca2	-s13 -i1.0 -u1.0 -l3 -p3
Vacation-Low	-n2 -q90 -u98 -r16384 -t4096
Vacation-High	-n4 -q60 -u90 -r16384 -t4096
Lee	75x75 Grid, 320 routes
Bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2

Table 8.2: Benchmark applications and their inputs used for evaluating SnCTM

The input configurations used for each benchmark are shown in Table 8.2. All the STAMP applications used their standard inputs [74]. Evaluations are made on the parallel region of the applications.

8.2.2 Performance

Figure 8.4 shows the performance improvement of SnCTM over the baseline architecture. In the legend, 1024 refers to the case where the size of the signature of both baseline and SnCTM is 1024 bits and 2048 represents when both systems use signatures of 2048 bits. In all the figures, the X axis represents the number of processors used for the experiment (2P→2 processors, 4P→4 processors and so on). The first observation that can be made from the figure is that SnCTM outperforms the baseline in almost all the cases with an average of 1.51X (1024) and 1.23X (2048). The performance improvement varies from 1X (Ssca2, 2P) to 4.62X (Vacation-High, 16P) for the signatures of 1024 bit width. In the case of signatures of 2048 bit, the performance improvement varies from 0.99X (Kmeans-High, 16P) to 2.93X (Vacation-High, 16P). The second observation from the figure is that the performance improvement over baseline increases as the number of processors is increased.

Improvements reported for both Kmeans applications is quite low in comparison to others. This is because the transactions used in those applications are small, hence the signature occupancy of the baseline is kept at a lower level as well. This leads to a reduction in false aborts caused by higher signature occupancy. The rest of the applications show moderate to higher performance improvements. Also it can be seen that the relative improvements of some applications is higher for 1024 bit signature than the 2048 bit signature. This is because when the signature size is smaller, the percentage occupancy increases. Also it increases the number of mappings that are destined for the same bit locations. Both these facts increase the number of false positives, which eventually degrades the performance. However this does not affect the performance of a SnCTM processor as much as it does for the baseline, due to the fact that signatures are used in the former only when necessary.

An observation that requires a further explanation in Figure 8.4 is the behaviour of Ssca2. This application is categorised as one having smaller transactions [74], hence the signature experiments presented in the literature have identified this as one with less sensitivity to signatures [19], similar to Kmeans. However, in Figure 8.4, the Ssca2 application has shown comparably significant improvement for the 16 core set for both signature configurations unlike Kmeans. The reason is, in the original Ssca2 application, most of the computation is performed outside transactions. In the current experiment, to ensure cache coherence is maintained for these computations, extra transactions have been added in those places. These were medium to large scale transactions. This makes Ssca2 exhibit a behaviour different to the ones reported in the

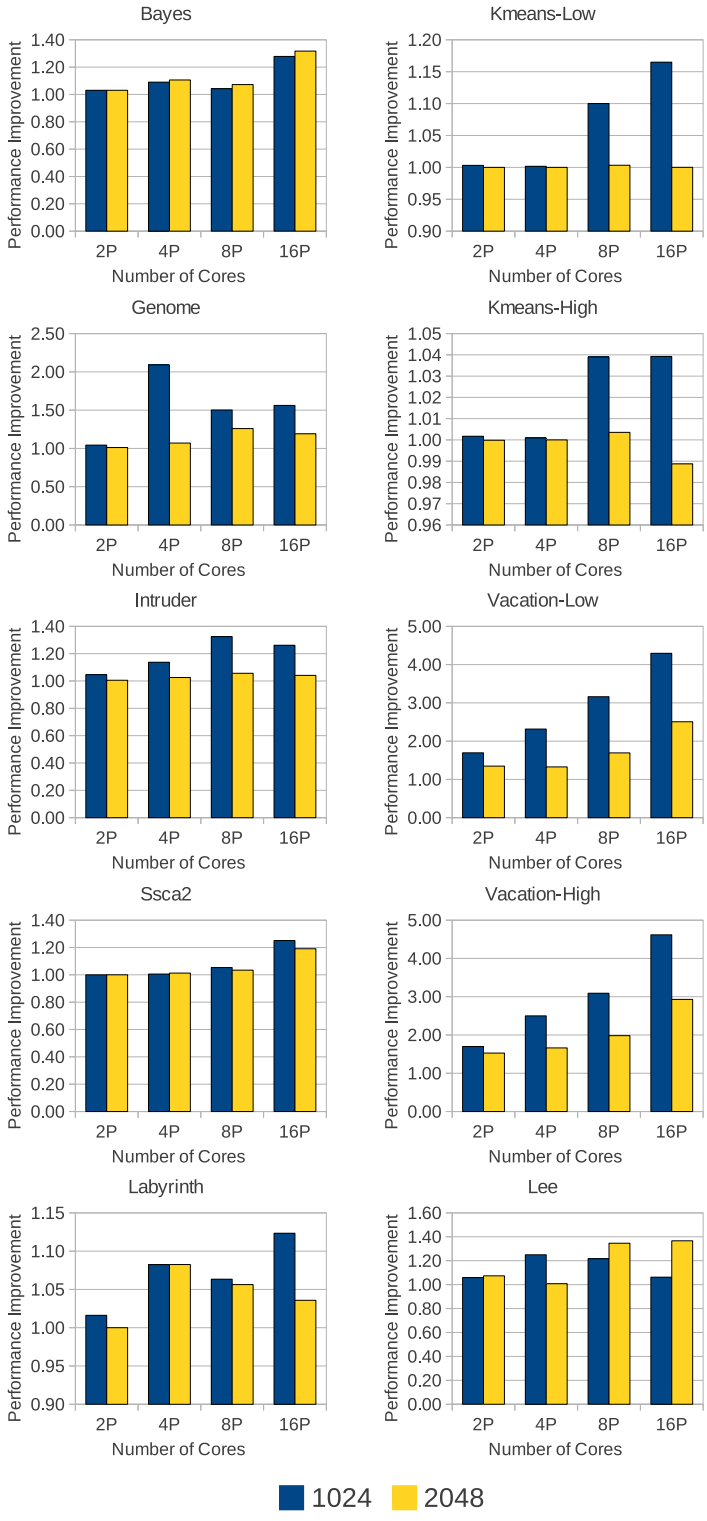

Figure 8.4: Performance improvement of SnCTM over the baseline

signature literature [19].

Table 8.3 shows the average performance improvement for each hardware configuration. There it can be seen that, as the number of processors increases, the performance improvement increases. In addition, the baseline performs better with a 2048 bit signature than with 1024.

Processors	1024	2048
2P	1.16	1.10
4P	1.45	1.13
8P	1.56	1.25
16P	1.87	1.46

Table 8.3: Average performance improvement of SnCTM over baseline

As the number of processors increase, bus contention also increases. This can be aggravated by introducing false positives to the system. The number of false positives produced in SnCTM is less than those produced in the baseline. False positives cause a processor to flush its cache and bring data from the next level memory. Even though the operations involved in this process are the same for any processor configuration, the effect on the bus contention increases rapidly as the processor count increases. Therefore the relative bus contention of the baseline with a higher number of processors is higher than the that of the baseline with a lower number of processors. Also when the number of processors increases, the number of live signatures increases as well. This increases the candidates for producing false positives. All this causes the performance improvement of SnCTM over the baseline with a higher number of processors to have a higher value than the improvement shown over a lower number of processors.

When the size of a signature is increased, the probability of two memory locations mapping to the same bits decreases. Therefore the accuracy of a signature increases as the size of it increases. This makes the baseline perform better with a signature of 2048 bit size. However, SnCTM still performs better than the baseline.

8.2.3 Characterization of SnCTM

Since SnCTM is based on the principle of adaptively changing the source used for detecting conflicts, first the effect of this on the number of transactions being aborted is analysed. Figure 8.5 shows the number of transactions aborted when a system has a 1024 bit signature. For comparison purposes the same figure also has the number of false positives occurring in both systems. The same description applies to Figure 8.6,

but those results are from a system with a 2048 bit signature. In all the figures, the X axis represents the number of processors used for the experiment (2P→2 processors, 4P→4 processors and so on).

From figure 8.5, it can be seen that in most applications SnCTM has encountered significantly less number of aborts than the baseline. The average lowest reduction in aborts is reported from Labyrinth, which is 14%. Bayes and Intruder have reported average reductions of 31% and 32% respectively. Reductions reported in both Kmeans applications varied significantly for low and high processor counts. The reduction of aborts in Kmeans-Low for lower processor count is around 20 % whilst that for higher processor count is around 80%. The same behaviour applies for Kmeans-High, reporting 10% and 42% reductions for low and high processor configurations. A similar behaviour is also reported in Ssca2, which has encountered 0% reduction for a 2 processor configuration, whilst having a 25% reduction for a 16 processor configuration. Lee also showed a 52% reduction in aborts compared to the baseline. Finally the highest number of abort reductions is reported in both versions of Vacation, reporting an average of 92% (Vacation-Low) and 88% (Vacation-High).

The number of transactions aborted is mainly a characteristic of the application and also it depends on the the contention management policy used in the TM system. That said, false transaction aborts which can occur from cache line sharing or false positives in signatures, can count towards this as well. Again looking at Figure 8.5, it can be seen that in some applications most of the aborts are encountered from false positives. The number of false positives occurring in the baseline system is measured against the number of aborts incurred. In Bayes around 71% of the aborts occur due to false positives. On average, the 77% of aborts incurred in Lee baseline are due false positives. For Genome this is around 89% and for both Vacation applications the figure goes to 99%. Moderate percentages are shown in Intruder (46%), Kmeans-Low (42%), Kmeans-High (24%), Labyrinth (26%) and Ssca2 (58%).

Similar to Figure 8.5, Figure 8.6 also shows that the number of aborts encountered in most of the applications is quite low when using SnCTM approach. However the percentage reduction has changed significantly in comparison to Figure 8.5. The lowest reduction of aborts is reported in both Kmeans applications, the baseline acquiring almost the same number of aborts as the SnCTM system. Among the other lower values, Intruder has 6% reduction whilst Labyrinth, Ssca2 and Bayes have 11%, 12% and 14% reduction of aborts when using SnCTM approach. Similar to the 1024 signature scenario, both Vacation applications report the most significant difference between the

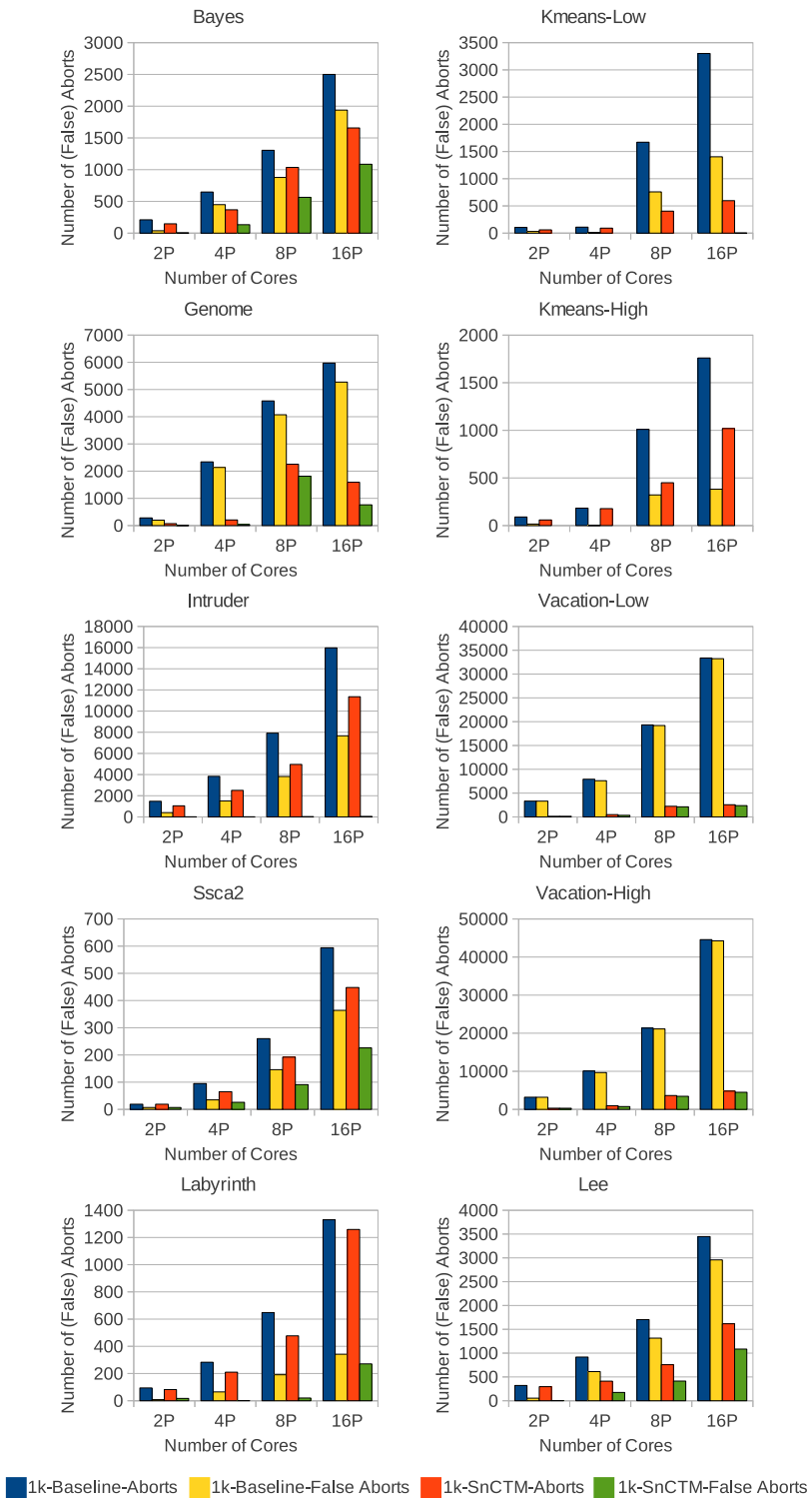

Figure 8.5: Number of aborts and false aborts occurred in both SnCTM and baseline with a 1024 bit signature

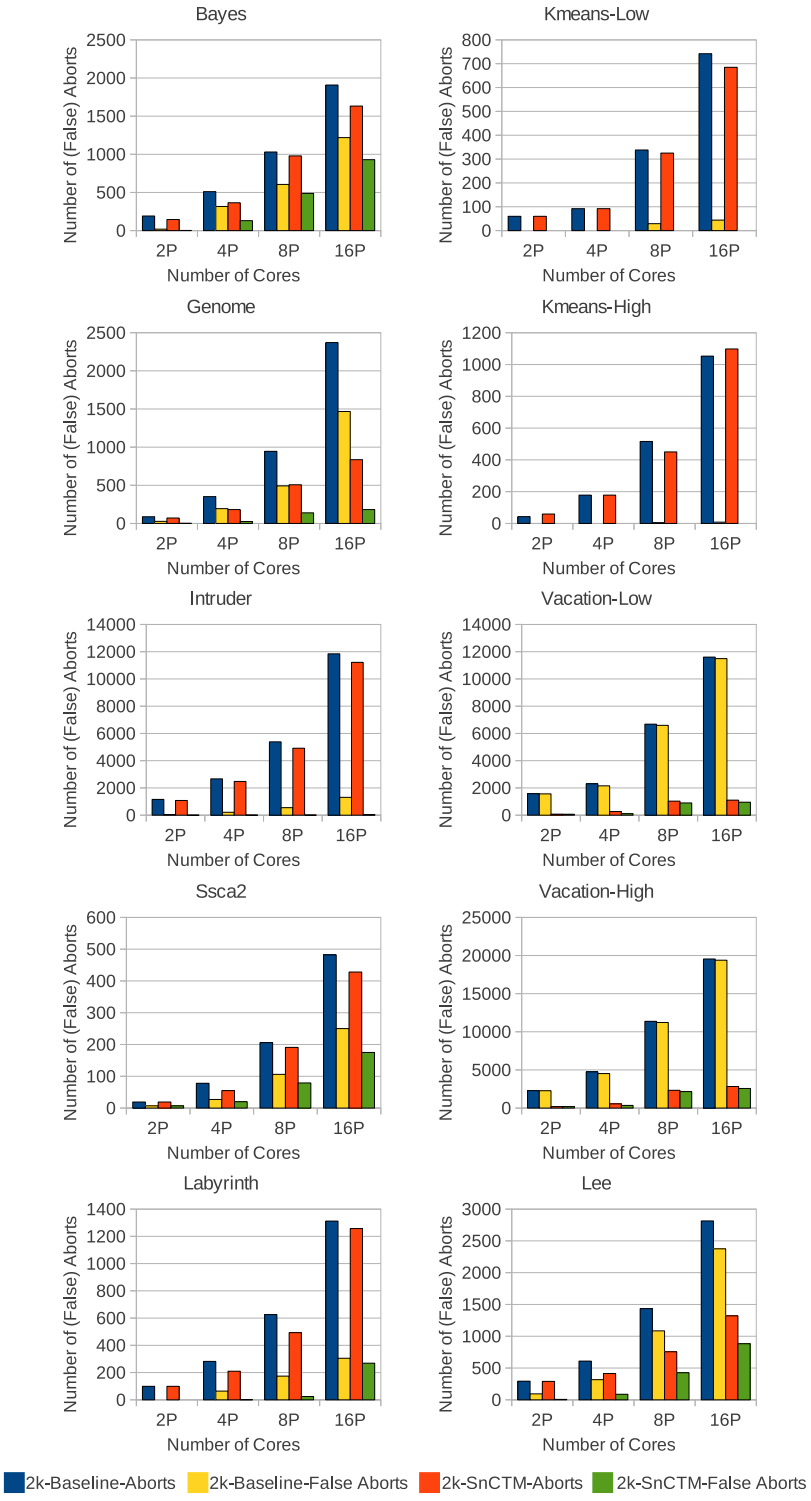

Figure 8.6: Number of aborts and false aborts occurred in both SnCTM and baseline with a 2048 bit signature

number of aborts occurring in baseline and SnCTM system. Vacation-Low reports a reduction of 89% of aborts when using SnCTM whilst Vacation-High achieves a reduction of 84%.

Increasing the signature length increases the accuracy of the conflict detection mechanism. Therefore when comparing the number of aborts occurring in the baseline with both signature configurations, as expected, the system with 2048 bit signature performs better. In Bayes, the 2048 signature results in 22% less aborts than the 1024 signature. Intruder (28%), Ssca2 (19%) and Lee (19%) also reported similar values. The increase of the signature has a moderate effect on Vacation-High (reporting a reduction of 52%) and Kmeans-High (reporting a reduction of 41%). Genome, Kmeans-Low and Vacation-Low have achieved a significant advantage from the increase of the signature length, by producing 81% 86% and 65% less aborts, respectively. Labyrinth has shown less sensitivity to the change in the signature length by having an almost zero reduction in number of aborts.

Having seen the number of aborts and false aborts which occurred in both systems for both signature configurations, discussion in now focused on analysing the number of false aborts occurring in both systems for both configurations. Figure 8.7 shows the number of false positives which occurred in both SnCTM and baseline systems for both signature configurations. In addition to 1024 and 2048 signature lengths, a signature configuration of 8k bits is also used for this experiment. The 8k signature is considered as a "perfect" signature and the aim is to compare the number of false positives occurring in both baseline and SnCTM systems with a perfect system. In the legend *1k* corresponds to a system with 1024 bit signature and *2k* corresponds to a system with 2048 bit signature. As the name suggests *Baseline* represents the baseline architecture and *SnCTM* represents the SnCTM architecture. In all the figures, the X axis represents the number of processors used for the experiment (2P→2 processors, 4P→4 processors and so on).

From Figure 8.7 it can be seen that, both signature configurations of SnCTM report less false positives (except Labyrinth 2P, 1k signature) than their corresponding baseline systems. It can also be observed that in certain cases the number of false positives in SnCTM is similar to that of the perfect system. Considering the applications individually, no false aborts occurred in both Kmeans application for both signature configurations in the SnCTM architecture and also in the baseline with a perfect signature. A significant difference of false positives between baseline and SnCTM can be observed in Intruder. This is because these applications (Kmeans-Low, Kmeans-High

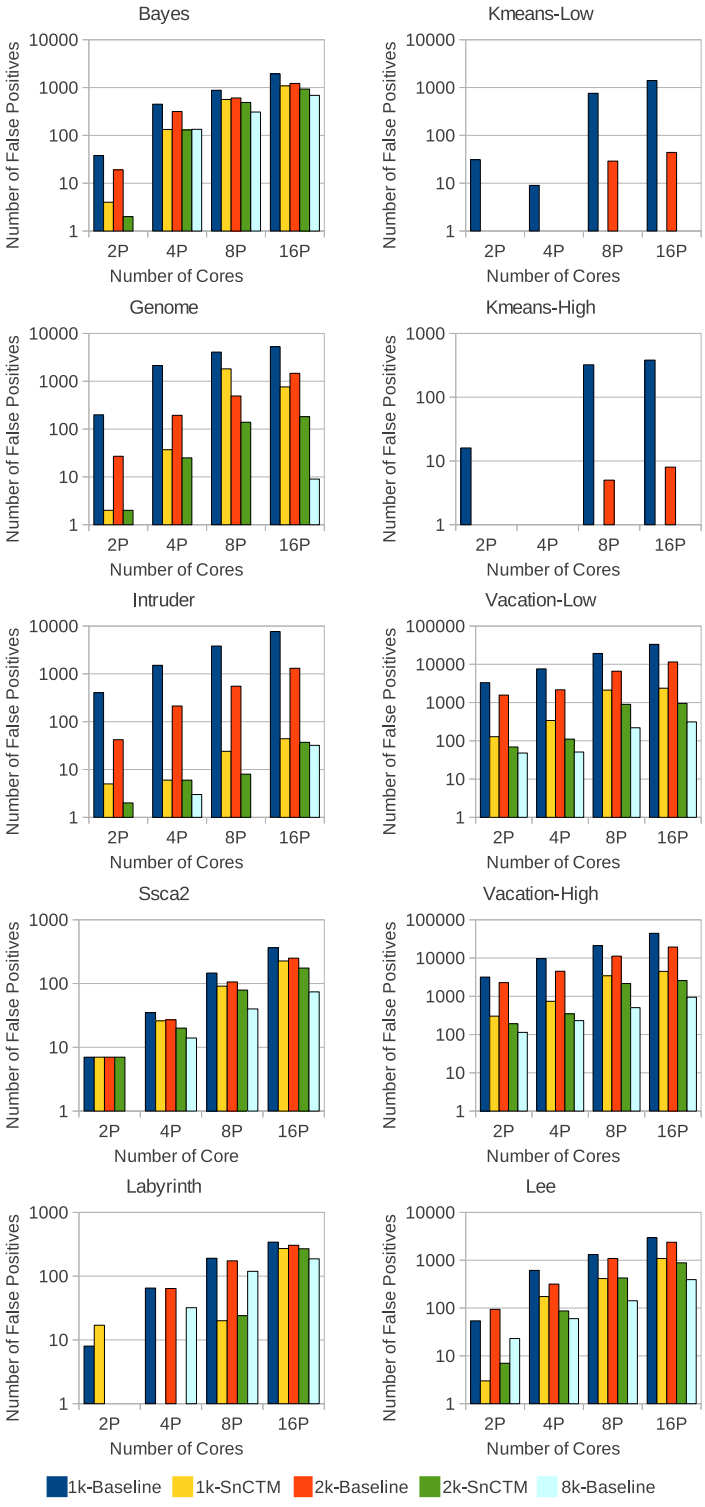

Figure 8.7: Number of false aborts occurred in both SnCTM and baseline with 1k, 2k and a perfect (8k) signature

and Intruder) have shorter transaction length, thus can fit in the L1 cache in most of the time. Therefore SnCTM can use cache lines to detect conflicts in most of the time, whereas baseline has to use signatures all the time. In the case of Ssca2, which is also categorised as an application having shorter transaction length in STAMP suite [74], does not show the similar behaviour in Figure 8.7. This is because, extra transactions have been inserted to the Ssca2 application to maintain coherence.

Both SnCTM configurations report higher false positives than the perfect system (still less than baseline) in Vacation-low, Vacation-high and Genome. All these applications have medium transaction length and low contention. In these applications not all the transactions fit in the L1 cache. Therefore SnCTM has to use signatures for some transactions, which causes it to produce some false aborts. However, SnCTM manages to keep the number of false positives lower than the baseline by adaptively changing source used during conflict detection.

In the cases of Lee, Labyrinth and Bayes all of them have longer transaction length. Therefore SnCTM also has to use signatures for most of its conflict detection, thereby increasing the false positives. However SnCTM is still able to produce less false positives than the baseline architecture.

One behaviour that can be observed in Figure 8.7 is that regardless of the underlying architecture, any signature implementation tend to produce more aborts when the number of processors increase. False positives are mainly caused by signature pollution (higher occupancy) and address aliasing. If the length of a signature is kept constant, the effect of the pollution will be the same for all the processor configurations. When the number of processors increases, the distribution of transactions among processors changes. For example if Transactions T1 and T2 were executed in processor 2, in a 2-core configuration, in a 4-core configuration, T2 will be executed in processor 3 concurrently with the execution of T1 in processor 2. If the addresses used in T1 and T2 have the tendency to produce similar hashes, then it is possible to introduce a new false conflict to the 4-core system which was not available in the 2-core system. Therefore it can be seen from Figure 8.7 that the number of false positives have increased with the increase of the processors in the system.

8.2.4 Sensitivity Analysis

This section presents a sensitivity analysis of the signature length in both baseline and SnCTM architectures. Signatures of size 64, 128, 256, 512, 1024, 2048 and 4096 bits are used for the experiment and an 8k bit signature is used as the perfect system.

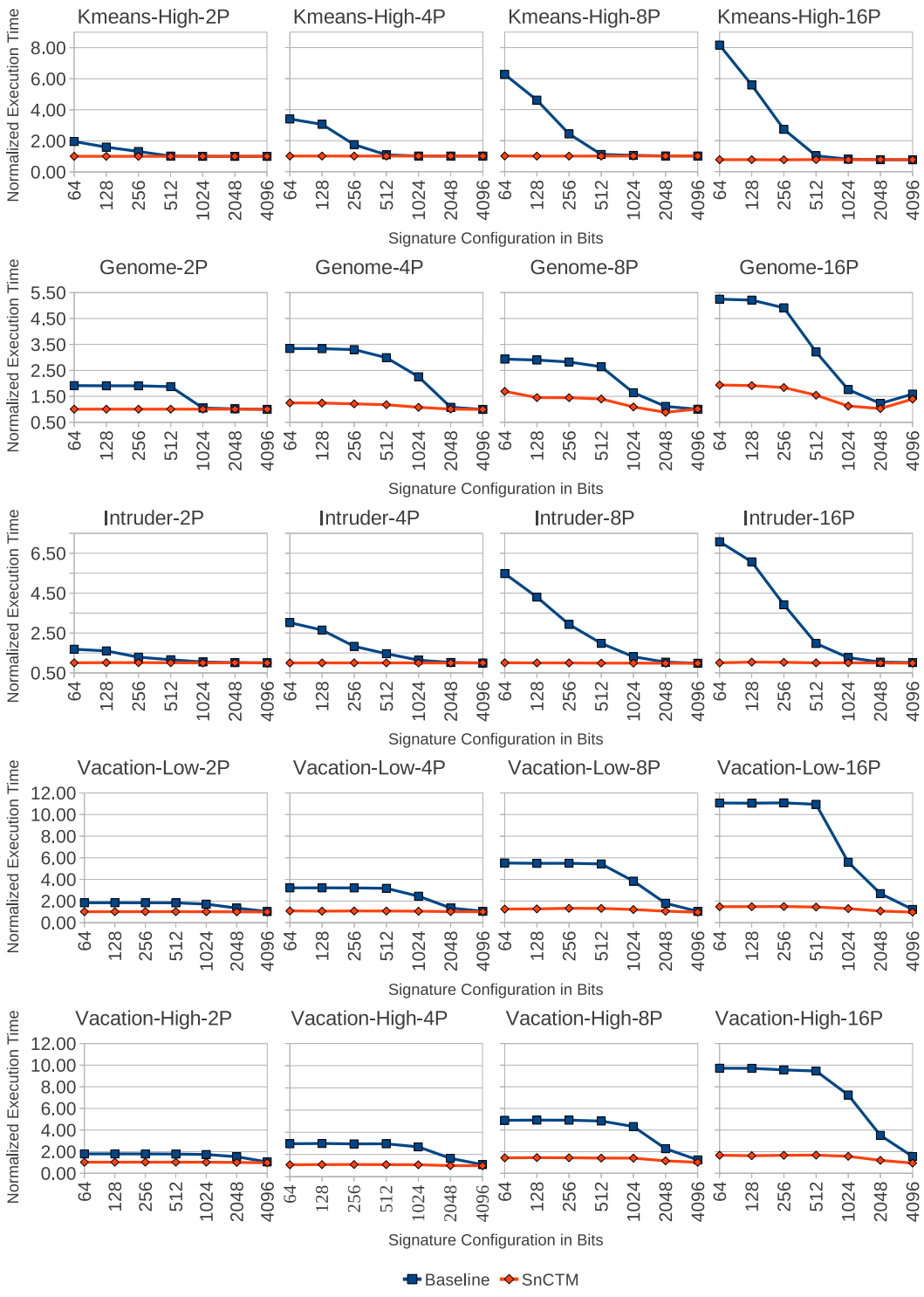

Figure 8.8: Signature sensitivity of baseline and SnCTM - Part I (execution time is normalised to the perfect signature)

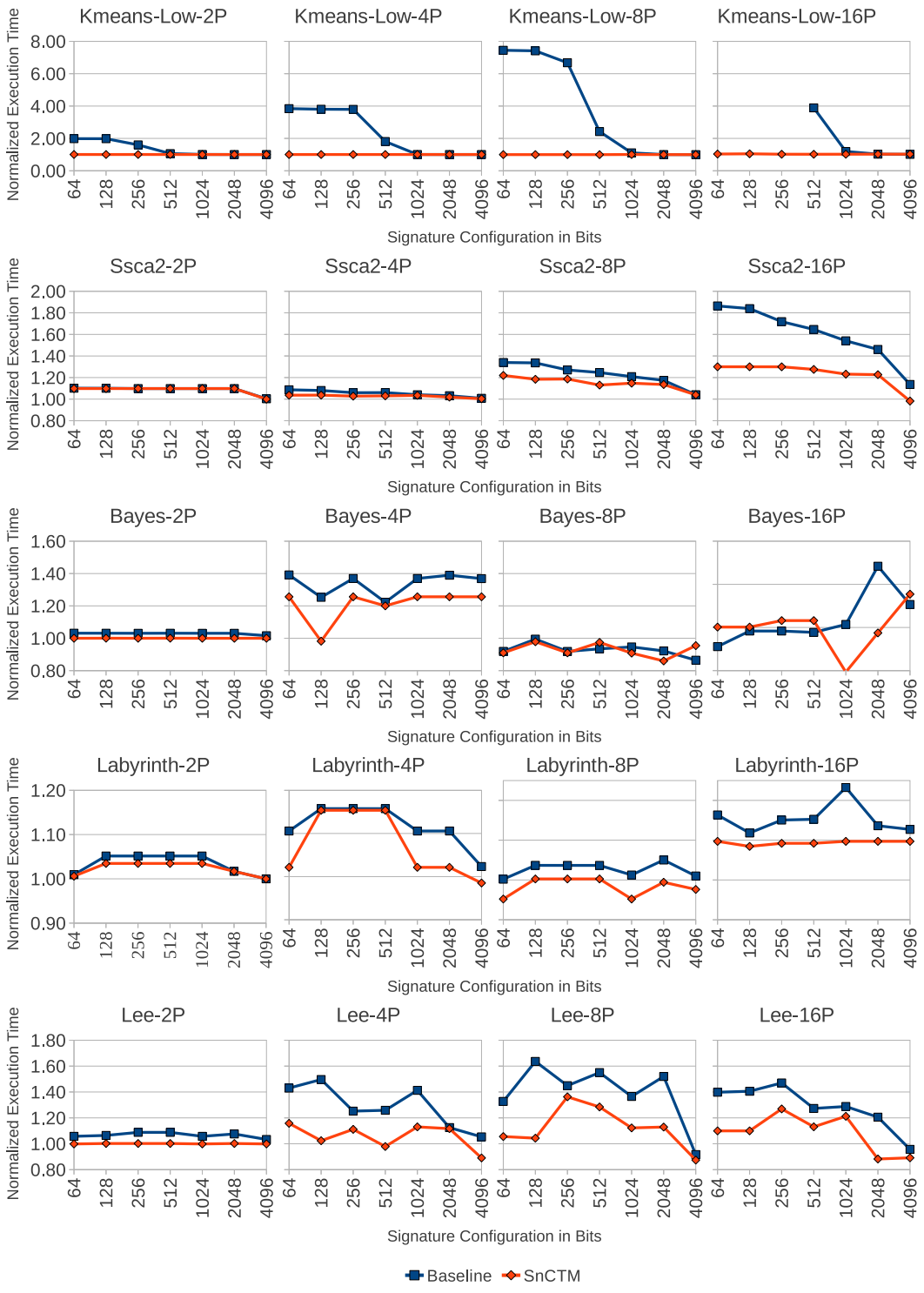

Figure 8.9: Signature sensitivity of baseline and SnCTM - Part II (execution time is normalised to the perfect signature)

The execution times of both baseline and SnCTM systems, normalized to the perfect baseline, are shown in Figures 8.8 and 8.9. The figure is divided into two parts in order to enhance the readability. For each signature configuration, experiments are carried out with multi-core processors having 2, 4, 8 and 16 cores. The Kmeans-low application did not successfully complete in a 16 core configuration with 64, 128 and 256 bits as the signature length. As can be seen from Figure 8.9, the application tends to produce a higher number of aborts, in the 16 core configuration, hence requiring quite of lot of restarting. The inability to complete the execution could therefore be due to a limitation in the simulator.

From Figures 8.8 and 8.9 two behaviours can be spotted in all the applications except Bayes, Labyrinth and Lee. The first behaviour is, in each processor configuration, the normalised execution time increases as the signature size reduces. However, in SnCTM this increase is either negligible or quite low in comparison to that of the baseline. The reason is, as the size of the signature decreases, the probability of producing false positives increases [95]. However SnCTM only uses signatures for situations where a cache overflow happens within the transaction. According to Figure 7.1, shown in Chapter 7 (Section 7.2), not many transactions fall into this category. Therefore the reduction in the signature size does not affect SnCTM as much as it affects the baseline. This means with the SnCTM approach, even a smaller signature can be used without compromising the performance of the system.

The second behaviour that can be spotted from Figures 8.8 and 8.9 is that, for all the applications except Bayes, Labyrinth and Lee, the normalised execution time of the baseline increases significantly in comparison to the perfect system, as the number of processors increases. Several reasons can cause this behaviour. The first is described in Section 8.2.3. That is, when the number of processor increases, false positives can be introduced due to a greater number of transactions being distributed. Also when the number of processors is increased, this increases the number of signatures in the system. Increasing the number of signatures means increasing candidates for checking conflicts. In an environment where signatures are perfect this does not and should not cause problems. However when the size of the signature is reduced, the percentage occupancy is increased. When the percentage occupancy is increased, the pollution of the signature is also increased. This leads to more false aborts.

The increased contention could count towards this as well. When the number of processors in a system increases, the contention for the interconnect increases. When the contention increases, a processor has to wait longer to get access to the shared

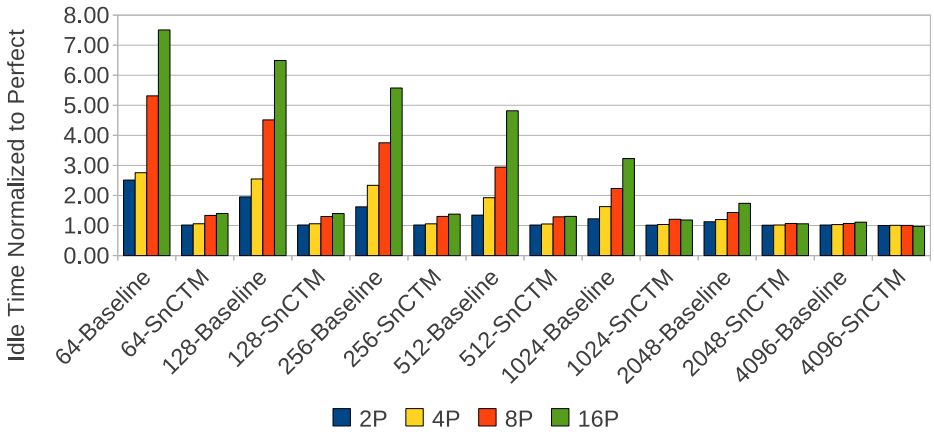

Figure 8.10: Idle time of SnCTM and baseline normalised to perfect

resources, thus the idle time increases. When processor idle time increases, transaction length increases. When a transaction takes a longer time to finish, it becomes more susceptible to abort. Aborted transactions result in flushing all the modified cache entries and bringing in all the data, once the transaction is restarted. This also increases the contention for the interconnect, thus making it a cyclic problem.

Processor idle time of both SnCTM and the baseline is measured for all the processor and signature configurations. Figure 8.10 shows the average of processor idle time for all benchmark applications (except Bayes, Labyrinth and Lee) for each hardware configuration. All the values are normalised to the perfect system. In the legend, 2P represents 2 processor configuration, 4P represents 4 processor configuration and so on. From Figure 8.10 it can be seen that for lower signature sizes as the processor count increases, the idle time of the baseline architecture increases significantly, where as in SnCTM it remains closer to the perfect system. Therefore the execution time of SnCTM remains comparable to a perfect system even with a smaller signature, whilst the baseline suffers a significant performance degradation.

In the case of Bayes, Labyrinth and Lee applications, firstly, they have a high contention [74]. This increases the number of aborts produced. Secondly, all these applications have longer transactions. Therefore it is very likely that they overflow during atomic execution, requiring the use of signatures to detect conflicts. This makes these these applications: (1) to have a lower performance improvement as the processor count increases; (2) to have less sensitivity to the signature length. Therefore they do not follow the same behaviour as others.

8.3 Summary

The first half of the chapter describes how to extend an existing HTM system to support the SnCTM concept. The second half of the chapter focuses on the evaluation of the SnCTM concept. First the evaluation is carried out with two commonly used signature configurations to identify the effect on execution time when using the SnCTM approach. The results are characterised to validate the hypothesis of Part II of the thesis. Thereafter a sensitivity analysis is performed to evaluate the effect of signature length on the execution time in both SnCTM and baseline. There it showed that a SnCTM system with a smaller signature can perform as well as a perfect signature whilst a baseline with the same signature size can suffer a significant performance degradation.

Part III

TM_EXIT: Exiting a Transaction In the Context of Hardware Transactional Memory

Chapter 9

TM_EXIT: A Case for Exiting a Transaction

Part III of this thesis makes a case for a functionality that allows a transaction to exit from a speculative region without committing it and start executing the statement immediately following the atomic block. The discussion is started by clarifying the ambiguity of two behaviours expressed for the specification of *Abort_Transaction*. Section 9.2 shows the need for having `TM_RESTART` functionality in the TM specification. The need for `TM_EXIT` functionality in TM is described using some code segments in Section 9.3. A preliminary experiment made to get an intuition for the need for such a functionality is described in Section 9.4. The proposed functionality, `TM_EXIT`, is presented in Section 9.5. The same Section also describes how to transform the code segments used in Section 9.3, to take the advantage of the proposed `TM_EXIT` function. Finally Section 9.6 summarises the chapter. Chapter 10 of presents the architectural requirements to support `TM_EXIT`. It also shows how to extend two baseline architectures to support the proposed `TM_EXIT` functionality. The advantages of using `TM_EXIT` in terms of performance is also presented in the same chapter. Transactional Memory approaches that could provide a functionality similar to `TM_EXIT` are discussed in Chapter 11.

9.1 Introduction

Transactional Memory(TM) [50] was initially proposed as a direct generalisation of the *load-linked-store-conditional* instruction, in order to provide atomicity to more than a single memory location. Since then, several HTM systems have been proposed

with different approaches for versioning and conflict detection (eg:TCC [39], Log-TM [77]). Several attempts [26, 73, 79, 99] have also been made to standardise the syntax and semantics of TM. Recently, several chip manufactures have unveiled proposals for hardware assisted transactional memory (eg: Sun‘s Rock processor [18, 29], Azul [25], AMD-ASF [1] and Intel's Haswell [55])).

Despite all these efforts, there is a mismatch in the specification of the *Abort_Transaction* function. When used at a time of a transactional conflict, the proposed behaviour is to discard all the speculative operations and reinstate the processor to the state that it was in at the beginning of the transaction. All the TM community agree on that. However when this functionality is invoked from the user code explicitly, two types of operations have been proposed. One type (eg: Log-TM [77]) proposes to discard all the speculative operations and to restore the state as it was at the beginning of the transaction. The majority of the TM proposals follow this approach. The other type of operation (eg: [79, 99]) proposes to discard all the speculative operations and to transfer the control to the end of the atomic block. So far, all the proposed hardware TM systems seem to follow the first specification.

This chapter argues that both functionalities are necessary for a hardware TM. In order to clarify the discussion, definitions for first and second type of operations need to be established. The first type of operation is named as `TM_RESTART` and the second type is named as `TM_EXIT`. The definition follows,

`TM_RESTART`: *Discard all the operations performed within the atomic section and restart the transaction.*

`TM_EXIT`: *Discard all the operations performed within the atomic section and transfer the control to the end of the atomic region.*

The following contributions are made in the Part III of this thesis.

- A successful case is made for supporting `TM_EXIT` functionality in HTM.
- In addition to extending the existing code segments to use `TM_EXIT`, a case has been presented where the expressiveness can be increased using the proposed functionality.
- As the third contribution, the feasibility of integrating `TM_EXIT` to two baseline HTM systems is presented and the proposed implementation is discussed.
- As the final contribution, performance evaluations of `TM_EXIT` on two HTMs are presented.

9.2 Motivation for TM_RESTART

This section argues the need for having TM_RESTART functionality in TM applications. In general, programmers place checks (eg: assert in C language) in a program to ensure that certain operations are performed correctly so that the intended outcome is produced. Similarly, within an atomic block, a programmer performs certain checks to ensure that functions invoked within the block provide the intended outcome. For example consider the code segment shown in Figure 9.1, taken from the Vacation application of the STAMP benchmark suite [74]. There, the function manager_addCustomer is supposed to insert the customerId, which is passed as an argument to the function, to a data structure inside the manager_t structure. The insert method of this data structure returns FALSE if the insertion is not successful. In such a situation a programmer may want to retry the transaction. Therefore the outcome of the insertion is checked and TM_RESTART is invoked to restart the transaction as shown in Figure 9.1. (Please note that only a portion of the transaction is shown.)

```
bool_t
manager_addCustomer (TM_ARGDECL  manager_t* managerPtr, long customerId)
{
    customer_t* customerPtr;
    ...
    customerPtr = CUSTOMER_ALLOC(customerId);
    assert(customerPtr != NULL);
    status = TMMAP_INSERT(managerPtr->customerTablePtr, customerId, customerPtr);
    if (status == FALSE) {
        TM_RESTART();
    }
    ...
    return TRUE;
}
```

Figure 9.1: TM_RESTART function used in Vacation application

Sometimes TM programmers use certain optimizations to reduce the number of transaction aborts. The Labyrinth application of the STAMP benchmark suite can be taken as an example of such a scenario. The objective of the program is to find paths from given source nodes to given destination nodes and to mark them in a shared global grid. In the algorithm proposed by Minh *et al.* [74], first the global grid is copied to a local data structure in each thread. Thereafter each thread finds the routes using this local grid. After the global grid is copied, the cache lines which are related to global grid are removed from the read set of the transaction using the *Early Release* [49, 100] feature. Using this approach the authors intend to reduce the transaction aborts caused by read/write accesses to the shared grid by multiple threads.

In the program, once a path is found it is marked in the global grid. During this marking, points used for the proposed path in the global grid are checked again to see whether they are already being marked by another thread. This is required because, finding the path was done on a local grid which is a snapshot of the global grid taken at the beginning of the exploration process, hence may not be up to date. Therefore, if a point in the proposed path is already taken by another thread, then the programmer may want to retry the transaction because this does not mean that there cannot exist a path from the given source to destination. This particular situation is shown in Figure 9.2. There, when a path is being added to the `grid_t` structure in the `TMgrid_addPath` function, it checks whether the location is empty or not. If the location is already taken, the transaction is restarted by invoking `TM_RESTART`. (Please note that only a portion of the transaction is shown.)

```
void
TMgrid_addPath (TM_ARGDECL  grid_t* gridPtr, vector_t* pointVectorPtr)
{
    ...
    for (i = 1; i < (n-1); i++) {
        long* gridPointPtr = (long*)vector_at(pointVectorPtr, i);
        long value = (long)(*gridPointPtr);
        if (value != GRID_POINT_EMPTY) {
            TM_RESTART();
        }
        ...
    }
}
```

Figure 9.2: `TM_RESTART` function used in Labyrinth application

This section showed that the `TM_RESTART` function is required in certain cases. Most of the hardware and software TM proposals provide this functionality, thereby allowing a programmer to retry the same transaction.

9.3 Motivation for TM_EXIT

This section argues the need for having `TM_EXIT` functionality. In TM applications, optimistic concurrency is maintained, for critical regions marked by programmers. The marking of critical regions is made either with the `atomic{}` keyword or with the `TM_BEGIN` and `TM_END` pair. Once critical regions are marked in an application, the underlying hardware/software/hybrid TM system ensures that the Atomicity, Consistency and Isolation (ACI) properties are maintained for the marked regions. In a lazy-lazy TM system, all the operations are performed speculatively within the critical section

and atomically committed at the end. In a hardware TM system, this commit phase involves writing all the modified cache entries to the next level memory.

However the usefulness of the speculative operations are not considered at the time of committing. This is because at the hardware level, this usefulness cannot be determined. The only information available is a set of memory locations and their prospective values. On the other hand, at the programming language level, the usefulness of a transaction can easily be extracted. Before continuing the discussion further, *usefulness* of a transaction needs to defined. The definition follows,

if an application contains a set of tasks and committing a transaction helps to reduce the size of this set, then it is a useful transaction.

Some examples are taken from (a) a benchmark, (b) a micro-benchmark and (c) a real-life scenario, to emphasise the need for `TM_EXIT` functionality.

9.3.1 Lee-TM [108]

This is a routing algorithm whose objective is to find a path from a given source point to a given destination point. The TM algorithm proposed by Watson *et al.* [108] comprises two phases: `expand` and `backtrack`. The `expand` routine starts from a source point and expands in all directions until it reaches the destination. Once it reaches the destination, it starts traversing back until it reaches the source point, thereby finding the optimal path. The transaction in this algorithm, encompasses both the `expand` and the `backtrack` methods. If the `expand` method was able to reach its destination, it returns TRUE, else it returns FALSE. The pseudocode of the algorithm is shown in Figure 9.3.

```
TM_BEGIN;
bool isFound=expand();
if(isFound){
   backtrack();
}
TM_END;
```

Figure 9.3: Lee-TM pseudocode

When the above definition of *usefulness* is applied to this scenario, the set of tasks are to find paths from different source points to different destination points. If a committing transaction is to be considered useful, it should have found a path from the given source to destination. Finding a path involves executing `expand` and `backtrack`

phases and the latter is executed only if the former returns TRUE. It is not guaranteed that `expand` always returns TRUE, which in turn prevents the `backtrack` phase from being executed. A transaction cannot be considered useful unless both `expand` and `backtrack` functions have been executed. In such situations, even if a transaction does not do any useful work, the commit operation still takes place. The commit phase involves writing back the modifications made to the local variables and to the local grid in the `expand` phase. Even though such commits are not useful to the overall application, they still use the interconnect to communicate speculatively modified cached entries to the next level memory.

9.3.2 Red-Black Tree

The Red-Black tree is a data structure used in computer science. The major operations associated with it are search, insert and delete. Even though this particular data structure is used for the discussion, the situation can be applied to any application that interacts with databases. The insert and the delete operations should incorporate some sort of search facility within them. This is because, before inserting an item, it is required to find whether another item with the same key exists in the tree. Similarly, to perform a delete operation the item with the corresponding key needs to be found. A search operation can be defined to return TRUE, if it finds an item with the given key. Then the insert and the delete operations can be performed accordingly. In a TM version of the Red-Black tree, a transaction comprises either, the search and the insert or the search and the delete. The atomicity between the two methods cannot be broken. The pseudocode of the algorithm is shown in Figure 9.4.

Along with the definition of usefulness, an insert operation can be considered as useful only if it tries to insert an item that does not exist in the tree. In the case of delete, it becomes useful only if the item exists in the tree. This discussion applies to any application that uses a database for storing information. For example, imagine a customer trying to book a room in a hotel. First they will search the room prices and availability. If a suitable one is found they will reserve it. In a TM version, both `search` and `book` have to be within a single transaction and it becomes useful only if the customer completes the booking stage. However with the current approaches, regardless of the usefulness of a transaction, a commit operation takes place.

```
insert_item(key){
    TM_BEGIN;
    bool isFound=search(key);
    if(!isFound){
        insert(key);
    }
    TM_END;
}
delete_item(key){
    TM_BEGIN;
    bool isFound=search(key);
    if(isFound){
        delete(key);
    }
    TM_END;
}
```

Figure 9.4: Red-Black Tree TM pseudocode

9.3.3 Java Exceptions

No specific examples are used for this discussion as it can be applied to many scenarios. Consider a Java program that has a critical section and is susceptible to produce an exception. In the code, a `TM_BEGIN` instruction can be placed after the `try` keyword. In a hardware TM, any operation performed after this instruction is performed speculatively until the `TM_END` is executed. Therefore if `TM_END` is placed within the `try-catch` block, it does not get executed if an exception is thrown. Since the objective is to execute `TM_END` regardless of whether an exception happens or not, it is placed within the `finally` block. A pseudocode of the discussion is shown in Figure 9.5. In this scenario, if a transaction is to be considered as useful, it should not produce any exceptions. However regardless of the usefulness of the commit, the application asks the underlying TM system to perform it.

```
try{
    TM_BEGIN;
    ...
    ...
}catch(Exception e){}
finally{
    TM_END;
}
```

Figure 9.5: Java TM code with exceptions

9.4 Performance Impact

To get an intuition of the usefulness of commits, a preliminary experiment is made with a lazy-lazy HTM system, similar to TCC [39], using 2-16 cores. Lee-TM [108] and a TM version of Red-Black tree are used for the experiment. The application code is instrumented to check the usefulness of a transaction at the commit time. For example in Lee-TM, if the `backtrack` phase is not executed, it is considered as a non-useful commit. In Red-Black tree, trying to insert an already existing entry or trying to delete a non-existing entry are considered as non-useful commits. For Lee-TM, a grid of 75X75 with 320 routes to explore is used as program parameters. For Red-Black tree, a tree with 20000 entries and transactionally inserting/deleting 16000 items with 50% probability for each action is used as program parameters. Table 9.1 shows non-useful commits as a percentage of the total commits.

Processors	Lee-TM	Red-Black Tree
2	35%	49%
4	35%	49%
8	34%	49%
16	33%	49%

Table 9.1: Non-useful commits

Table 9.1 presents two interesting facts. One is that, within the application program itself it is possible to determine whether a commit is useful or not. The other observation is that quite a number of commits are not useful for the overall program completion, in the applications used for the study. If the underlying TM is not notified about these non-useful transactions, a commit phase will take place for those transactions similar to others. In HTM, a commit phase involves communicating the information about speculative modifications (write-set) to other processors and updating the next level memory. Both of these operations require to use the interconnect, which is a shared resource in a multi-core processor. Increased usage of shared resources increases the contention for them. This increased usage could be reduced if these non-useful commits were avoided. Therefore a functionality is required for a TM API to notify the underlying TM, that committing the modifications made in current transaction does not do any useful work, therefore proceed to the instruction following the atomic block.

The `TM_EXIT` functionality defined earlier in this chapter fits well for this purpose.

First of all it increases the programmability of TM programming, allowing a programmer to have multiple exit points in an atomic block. Secondly it may also improve the performance of a hardware TM system, by not committing the non-useful data. However this functionality has not yet been integrated to a hardware TM system.

9.5 Defining and Using TM_EXIT

From a programmer's point of view, the purpose of the `TM_EXIT` functionality is to notify the underlying TM mechanism that the current transaction is a non-useful one, and request an exit from the atomic region. From system's point of view, this information can be interpreted as "*stop speculation from this point onwards*". However this is not sufficient for the underlying TM to function properly because to fulfil the requirements of `TM_EXIT` it needs to transfer the execution flow to the line immediately following the atomic block. In other words, `TM_EXIT` looks like a `jmp` instruction which transfer the execution flow to the line immediately following the atomic block. If used in this manner, the proposed functionality will become unattractive among the TM community, similar to the obsolete use of `goto` statement in C/C++. However if the `TM_EXIT` functionality is used alongside the definition of *usefulness*, such an explicit control transfer becomes unnecessary. Before analysing why this is the case, the discussion is directed to show how to use the `TM_EXIT` functionality alongside the definition of *usefulness*.

In order to use the `TM_EXIT` using this approach, first a usefulness criterion for a transaction needs to be established. Then it is necessary to check whether there are any program statements residing outside the usefulness criterion, that affect the usefulness of the application. If there are no such statements then the `TM_EXIT` function can be used. The rest of the section describes how to modify the code segments shown in Section 9.3, according to these two steps.

9.5.1 Integrating TM_EXIT to Existing Applications

In the case of Lee-TM, the usefulness criterion is whether the `expand` method is able to reach the destination. Then it is necessary to consider the effects that can be caused by any program statements outside the usefulness criterion. Write operations performed within the `expand` method and the write operation performed on the `isFound` variable fall in to this category. None of these operations affect the usefulness of the program.

In this application, only operations that could affect the usefulness are the ones performed within the `backtrack` method, which is already encompassed in the usefulness criterion. This can be seen in Figure 9.6(a). Therefore `TM_EXIT` functionality can be used in the Lee-TM application. The application code needs to be modified as shown in Figure 9.6(b) to use the proposed feature.

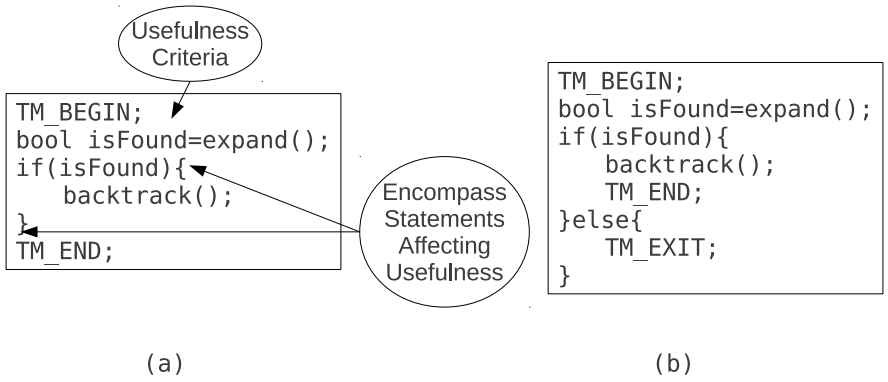

Figure 9.6: Modifying Lee-TM pseudocode to use `TM_EXIT`

In the case of the `insert_item` method of Red-Black Tree, the usefulness criterion is not finding an entry with the same key as the one which is currently being inserted. Similarly, for the `delete_item` method, the usefulness criterion is finding a matching entry with the key that is required to be deleted. Write operations performed on stack variables in the search method and the write operation performed on the `isFound` variable fall into the category of operations performed outside the usefulness criterion (shown in Figure 9.7(a)). These operations do not affect the usefulness of the program. The only operations that could affect this are the ones within the `insert` and the `delete` methods. Since they are already encompassed within the usefulness criterion, `TM_EXIT` functionality can be used in this situation as well. The pseudocode of the modified Red-Black Tree is shown in Figure 9.7(b).

In the case of Java exception code segment (Figure 9.5), neither usefulness criteria nor the statements affecting the usefulness can be defined as the code segment is a generic one. A simple assumption to make here is that, it is very unlikely for a transaction to be considered useful if it has encountered an exception. If an exception is raised, it will be caught by the relevant `catch` block in the program flow. `TM_EXIT` can be placed in all `catch` blocks allowing the execution to exit from the atomic block. This is shown in Figure 9.8. In addition to adding `TM_EXIT`, the program is slightly modified from the one shown in Figure 9.5. Since `TM_EXIT` is placed within the `catch` block, it is no longer required to place the `TM_END` in the `finally` block. As shown in Figure

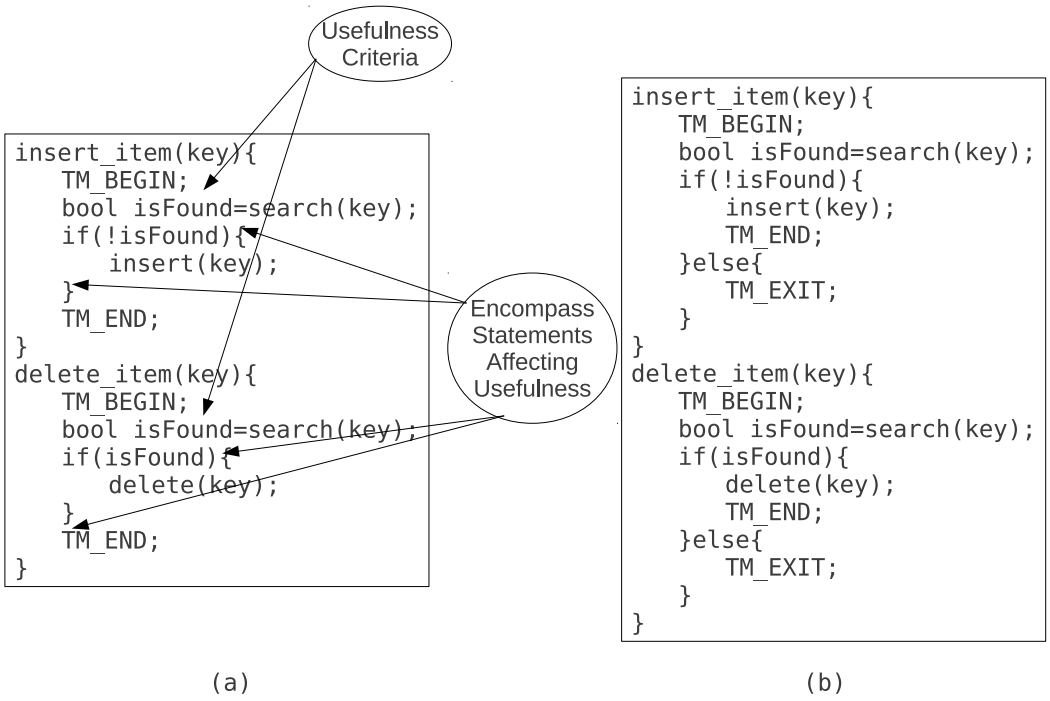

Figure 9.7: Modifying Red-Black Tree TM pseudocode to use `TM_EXIT`

9.5, `TM_END` is now placed within the `try-catch` block. If no exceptions occur during the execution of the `try-catch` block, `TM_END` is executed thereby committing all the speculatively modified data. If an exception has occurred it will be caught by the relevant `catch` block and `TM_EXIT` will be executed, thereby discarding the non-useful writes.

```
try{
    TM_BEGIN;
    ...
    ...
    TM_END;
}catch(Exception e){
    TM_EXIT;
}
```

Figure 9.8: Modified Java code to used `TM_EXIT`

9.5.2 Implicit Control Transfer with TM_EXIT

Having shown the usage of *usefulness* criteria to modify applications, the discussion is now focused on describing how the program execution flow is implicitly transferred to the line following the atomic block. Consider the code segment shown in Figure 9.9,

which is a numbered version of Lee-TM shown in Figure 9.6(b). For simplicity, assume each pseudocode is an instruction and the numbers are the memory addresses of those instructions. In general, the *Program Counter* (PC) register stores the address of the next instruction to be executed. Consider a case in which a path is found. When that is the case, the `expand` returns TRUE, then the program execution jumps to address 04, which is `backtrack`. When `backtrack` is being executed the PC register is pointing to address 05 which is the `TM_END`. When `TM_END` is being executed the PC register is pointing to address 09 which is the end of the atomic block.

```
01: TM_BEGIN;
02: bool isFound=expand();
03: if(isFound){
04:    backtrack();
05:    TM_END;
06: }else{
07:    TM_EXIT;
08: }
09:
```

Figure 9.9: Implicit control transfer in Lee-TM pseudocode

Now consider a case where a path cannot be found, which made `expand` return FALSE. When this happens, the program execution jumps to address 07, which is the `TM_EXIT`. When this instruction is being executed, the PC is pointing to the address 09, which is the end of atomic block. Therefore when the execution of `TM_EXIT` is completed, program execution will automatically be transferred to the line immediately following the atomic block without any extra effort.

9.5.3 Incorrect Usage of TM_EXIT

```
0: TM_BEGIN;
1: AAA
2: BBB
3: if(condition){
4:  CCC
5:  DDD
7: }
8: EEE
9: TM_END;
```

(a)

```
 0: TM_BEGIN;
 1: AAA
 2: BBB
 3: if(condition){
 4:  CCC
 5:  DDD
 6:  TM_END;
 7: }else{
 8:  TM_EXIT;
 9: }
10: EEE
```

(b)

Figure 9.10: Incorrect usage of `TM_EXIT`

The discussion of implicit control transfer raise the question "*What happens if there is an instruction outside the usefulness criterion, but after it ?*". For example consider the code segment shown in Figure 9.10(a). There, the condition in line 3 could easily be misunderstood as the usefulness criterion of the transaction. In such a situation the modified code may look similar to the one shown in Figure 9.10(b). This is clearly an incorrect usage of `TM_EXIT`. Recalling the two steps of using `TM_EXIT` are to, define the usefulness criterion and to ensure the operations outside the usefulness criterion does not affect the usefulness of the program. In this situation, the operation at line 8, clearly affects the usefulness of the application. This is because, regardless of the value of the `condition`, line 8 gets executed. Therefore line 8, itself contributes to the usefulness of the application. In this situation, both the *if-condition* in line 3 and the statement in line 8 collectively contribute to the usefulness of the application. Therefore, line 3 alone cannot be considered as the usefulness criterion, hence this application cannot use the `TM_EXIT` functionality.

9.5.4 Increasing Expressiveness With TM_EXIT

This section describes a situation where the expressiveness can be increased with `TM_EXIT`. Following is the problem statement.

A linked list, whose size is unknown, needs to be reversed if the size of the list is greater than a certain threshold.

A conventional and a naive way of solving the problem is to count the items in the list in a first pass and to reverse it in a second pass, if the size is greater than the threshold value. This is shown in Figure 9.11.

```
length = count(list);
if(length > THRESHOLD) {
   reverse(list)
}
```

Figure 9.11: Pseudocode showing the conventional approach

The ability of TM to operate speculatively within an user specified block, integrated with the proposed `TM_EXIT` can be used to provide a better solution for the above scenario. The intended behaviour of `TM_EXIT` is to discard all the speculative operations and to transfer the control to the line immediately following the atomic block. Therefore the `reverse` method can be modified to speculatively reverse the list while counting the elements. When the end of the list is reached, if it is found to have more items than the given threshold, the transaction is committed. If this is not the case, it

exits from the transaction using TM_EXIT, thereby discarding all the operations made to reverse the list. This is shown in Figure 9.12.

```
modified_reverse(list){
    TM_BEGIN;
    while(more elements) {
        count
        reverse
    }
    if(count > THRESHOLD){
        TM_END;
    }else{
        TM_EXIT;
    }
}
```

Figure 9.12: Pseudocode of revised reverse method using TM_EXIT

In the naive approach, shown in Figure 9.11, for lists whose length is greater than the threshold traversing is done twice (one for counting and one for reversing). In the new approach, shown in Figure 9.12, regardless of the size of the list, traversing is done only once. Therefore the amount of computation required to derive the solution is also reduced. This means TM_EXIT also has the potential to increase the performance of an application in certain situations, in addition to increasing the programmability by having multiple exit points. It can be noted here that, the two steps required to follow in order to use TM_EXIT (described in Section 9.5), have not been followed in this situation. This is because, here TM_EXIT is not being introduced to an existing atomic block, instead an atomic block is added together with TM_EXIT to a code segment to increase the expressiveness. Also the use of atomic blocks are not for maintaining synchronization, but for maintaining the speculative execution.

9.6 Summary

This chapter presented a case for having TM_EXIT functionality in TM programming. It started the discussion by first clearing the ambiguity of two behaviours expressed for the *Abort_Transaction* operation, by defining each type (TM_RESTART and TM_EXIT). Thereafter the need for both types of operations in TM, is presented using some known benchmark applications. The steps required when modifying existing applications to make use of TM_EXIT function is described using the same set of examples used for making the case for TM_EXIT. Finally it shown how to use the TM_EXIT function in order to increase the expressiveness.

Chapter 10

Implementation and Evaluation of TM_EXIT

This chapter describes the architectural support for TM_EXIT and the evaluation of it, in terms of performance. Section 10.1 describes two baseline TM systems and how to extend them to provide TM_EXIT functionality. Even though improving performance is not a major goal of the proposal, a study is done to evaluate the effect of adding TM_EXIT functionality to two baseline TM systems. This is presented in Section 10.2. Finally Section 10.3 summarises the chapter.

10.1 Architectural support for TM_EXIT

This section discusses how to extend two hardware TM systems to support TM_EXIT functionality. Two improved versions of Transactional Memory Coherence and Consistency (TCC) [39] are used as baseline architectures. Both baselines are similar to those described in Chapter 4, Section 4.3. For the sake of completeness, a brief description of each is given in Sections 10.1.2 and 10.1.3. The transactional memory implementation in the baselines is similar to any other lazy-lazy hardware TM system. When the TM_BEGIN instruction is executed, a flag (*IN_TX* in Figures 4.8 and 4.9) is set. When this flag is set, all the subsequent operations are performed speculatively until the TM_END instruction is executed. In order to provide an unbounded amount of transactional data, the baseline uses hardware signatures [95] to maintain the read and write sets, using parallel bloom filters to increase accuracy. Since the baseline architectures are based on TCC which does not implement any coherence protocols, transactions are used to maintain coherence and consistency as well. Therefore at the

end of a transaction, the next level memory copies are updated and local copies which are read/written are flushed. This is necessary to avoid local caches using stale data due to the fact that no conventional coherence protocols are used.

When a processor needs to commit a transaction, it first requests commit permission from a centralised *commit-arbiter*. Commit permission is granted based on a least recently granted policy. Once the commit permission is granted, the committing processor broadcasts its write-signature to all the other processors. Upon receiving this write-signature, each processor performs a bitwise AND operation on their read-signature. If all the hashes in the resulting signature are non-zero, then it is considered as a conflict and the processor aborts. After sending the write-signature to all the other processors, the committing processor updates the next level memory (either level 2 cache or main memory) with all the speculatively modified values. During this commit phase, the communication arbiter denies any request to use the interconnect. Once the next level memory is updated with all the speculatively modified cache entries, all these entries need to be flushed and both read and write signatures need to be cleared as well. The two baseline systems differ from each other, from the way they handle cache overflows within a transaction.

10.1.1 Requirements for TM_EXIT

When invoked from the user code, `TM_EXIT` is supposed to perform two operations. One is to stop performing speculatively and the other is to transfer the control to the line immediately following the atomic block. The first objective, that is to stop performing speculatively can be done by clearing the *IN_TX* flag. From the examples discussed in Chapter 9, Section 9.5.2, it is clear that no explicit operations are required in order to transfer the control to the line immediately following the atomic block. In that sense `TM_EXIT` is similar to a commit operation except the latter communicates the modifications made within the atomic region to the other processors. When `TM_EXIT` is executed, all the speculatively modified cache entries need to be cleared before executing the next instruction. In that sense, `TM_EXIT` is similar to an abort operation except the latter restores registers. To summarise, the operations associated with `TM_EXIT` are, to clear the *IN_TX* flag and to clear all the speculatively modified entries.

10.1.2 Baseline-1: TM-S

Overflows are serialised in the first baseline (TM-S) when addressing cache overflows within a transaction. That is, when a cache entry needs to be rejected while a processor is inside a transaction, permission is sought from the *overflow arbiter*. Overflow permission is granted based on a least recently granted policy. Once the overflow permission is granted, the processor flushes the cache line from its L1 cache and updates the corresponding entry either in L2 cache or main memory. A processor needs to ask for overflow permission only if the cache line is modified during the current transaction. An extra 'W' bit is used to mark all the speculatively modified entries. A dirty bit is not sufficient for this purpose because the entry could have been dirty due to a write operation performed outside a transaction. If the 'W' bit is not set, there is no need to seek overflow permission, a processor can flush the entry to its original location. If an overflow request is denied, the processor stalls until the request is granted. Even though the *commit-arbiter* operates on a *least-recently-granted* policy, in the TM-S baseline, there is an exception for processors which have been granted overflow permission. That is, all the commit requests from other processors are denied, until the overflowing processor commits. Due to this approach, the overflowing transaction becomes an unabortable one.

The TM-S baseline needs a mechanism to handle unabortable transactions. Imagine a situation where a processor which has been given overflow permission, invoked `TM_EXIT`. The associated operations are to discard all the speculative changes and to continue to the next instruction. In this baseline, the modifications related to the overflowed memory locations cannot be discarded as the original memory locations have been modified. When this is analysed from the perspective of the application code, two types can be observed. In one type, only the code region encompassed with the usefulness criterion modifies the global data. Examples like Lee-TM and Red-Black Tree (discussed in Chapter 9, Section 9.5.1) fall in to this category. In the other type, those similar to the Linked-list example (described in Chapter 9, Section 9.5.4), a usefulness criterion is not used when adding the `TM_EXIT` functionality.

In the first type, if the `TM_EXIT` functionality is not supported, a commit operation will take place. In the case of Lee-TM, for non-useful transactions, this involves updating the locations in the local grid. Similarly, for Red-Black tree this involves writing back local variables which are modified during the search operation. None of the other threads are interested in the modifications made to these locations. In terms of processors, none other than the one who initialised these are interested in these memory

locations. If a transaction cache overflow happens for these non-useful transactions, some of these memory locations will get modified before the commit phase and the rest will be updated during the commit phase. If the commit phase did not update the rest of the local grid, it would not make any difference as the grid is initialised at the beginning of the next transaction. Similarly all the local variables of the Red-Black tree will be initialised during the next search method. Therefore for the first type, reinstating the overflowed locations or committing the rest of the write set is not required for the correctness of the execution. However if a need arises, `TM_EXIT` can be configured to commit the rest of the speculative entries in case of a transaction cache overflow. To summarise, when `TM_EXIT` is invoked from the user code, if no transactional cache overflows have occurred all the speculatively modified cache entries are discarded. If there were transactional cache overflows, the system has the flexibility to either commit or discard the remaining cache entries. For the experiments the latter option is used, that is to discard the remaining speculative values in the level 1 cache.

In its current version, the TM-S baseline cannot be used to support `TM_EXIT` with applications of type two (like the Linked-List example described in Chapter 9, Section 9.5.4). This is because, if a transaction which has a cache overflow, decided to exit from the atomic block, then it could lead to an erroneous output. For example consider a situation where a cache overflow occurred during the speculative reversing of the list and this modified the original memory location. However the number of elements in the list is less than the threshold, therefore the transaction decided to exit using `TM_EXIT`. Now the original linked-list is modified even though it should remain unmodified. Therefore in order to ensure that consistency is not undermined for all types of applications, further work is required when integrating the `TM_EXIT` to TM-S baseline.

10.1.3 Baseline-2: TM-U

In order to support an unbounded amount of transactional data, the second baseline (TM-U) overflows to a separate uncached area of memory as in Large Transactional Memory (LTM) [3]. The design and the protocol are similar to those of LTM, except that TM-U does not stall to check for potential conflicts that might arise from overflowed locations, since it uses signatures. When a cache line with the dirty bit set is going to be overflowed, the entire cache line including all the tag, valid, dirty and data bits are preserved in this uncached area. Each entry is indexed by the hash value of the overflowed memory location. Each processor has an extra register (*Overflow_Address*)

which points to the starting location of this separate area. If more than one memory location produced the same index, a linked list is formed. Finding an entry involves first getting the index and then getting the corresponding cache entry or list of cache entries stored under that index. Then a linear search is performed by comparing the tag and index of each element in the list. TM-U has an extra bit called 'O' per cacheline to indicate the overflow status. This is set when a cache line is overflowed and is cleared only when a transaction commits or aborts. Even if an existing cache line is replaced with new data, this bit does not get changed.

When `TM_EXIT` functionality is invoked in TM-U baseline, in order to clear the speculatively modified entries it is only necessary to clear the modified cache entries and the entries in the overflow area of the memory. Since overflowing does not affect the original memory location, both types of application can be executed in TM-U baseline.

10.2 Evaluation

The evaluation of `TM_EXIT` functionality, in terms of performance, is presented in this section. After discussing the evaluation setup in Section 10.2.1, performance evaluation of `TM_EXIT` is presented in Section 10.2.2. In that section performance of TM-S and TM-U systems that support the `TM_EXIT` functionality is compared against TM-S and TM-U systems that do not support such a feature. Characterisation of the results are presented in Section 10.2.3. Section 10.2.4 describes the performance evaluation of `TM_EXIT` when used to improve the expressiveness.

10.2.1 Evaluation Setup

The simulation environment used for evaluating `TM_EXIT` is similar to the one used for evaluating DaCTM, described in Chapter 5, Section 5.3. For the sake of completeness, a brief description of the system is presented in this section, readers are directed to Chapter 5 for a more elaborative description.

Since the proposal relies on transactions, a lazy-lazy hardware transactional memory system is modelled in Simics [70], a full system simulator running Linux (version 2.6.16). The TM system is configured with the components shown in Table 10.1. In addition to those, the baseline-2 (TM-U) uses a perfect hash function to index its overflowed memory locations. Lee's routing algorithm [108] and a TM version of

the Red-Black tree are used to evaluate the TM_EXIT functionality. Both applications were modified to exit from a transaction if it is found to be a non-useful commit, as shown in Figures 9.6(b) and 9.7(b). Unmodified versions of both applications were executed for comparison purposes. Lee-TM uses 75X75 grid and 320 routes as the input. A tree with 20,000 (0↔200,000) nodes and 16000 (0↔200,000) insertions/deletions with 50% probability for each was used for the Red-Black tree experiment.

Component	Feature
Processors	1-16, in-order
L1 Data Cache	2 way assoc, 64 B line, 32 KB size, 2 cycle latency, private per core
Signature	2048 Bits, 4 Parallel H3 [12] Hash functions
L2 Data Cache	8 way assoc, 64 B line, 4 MB size, 20 cycle latency, shared
Interconnect	Split-transaction bus, 4 cycle latency, 64 B data width
Main Memory	100 cycle latency

Table 10.1: Components and features of the TM_EXIT evaluation environment

10.2.2 Performance

The performance improvement of using TM_EXIT over baseline architectures that do not support such functionality is shown in Figure 10.1. With the TM-S architecture, using TM_EXIT functionality, a maximum performance improvement of up to 1.35X has been achieved and with TM-U the maximum improvement went to 2.28X. From Figure 10.1, it can be seen that the Lee-TM application has taken more advantage over the Red-Black tree application by using TM_EXIT functionality to exit from non-useful transactions. Also it can be observed that, for the Lee-TM application, in the TM-U system the improvement over the baseline increase significantly as the number of processors increases. However, this does not apply to the TM-S system. Reasons causing this behaviour are discussed in Section 10.2.3.

10.2.3 Characterisation of TM_EXIT

The results of the performance evaluation are characterised with several parameters in order to find out the effect of adding the TM_EXIT functionality to two existing HTM systems and what makes the performance improvements vary between both systems

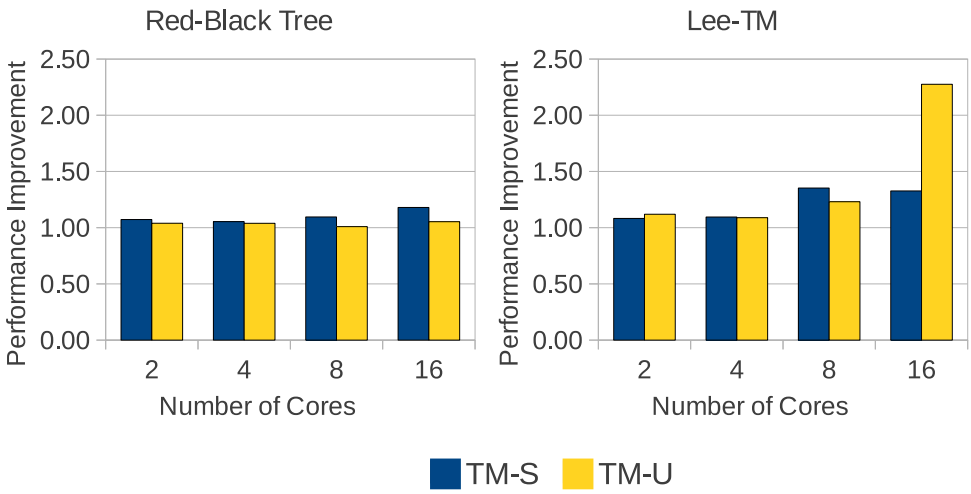

Figure 10.1: Performance improvement when using `TM_EXIT` over baseline

and applications. First, the percentage of transactions that executed the `TM_EXIT` instruction is measured. Table 10.2 shows how many times the `TM_EXIT` instruction has been executed as a percentage of the total number of commits. There it can be seen for the Red-Black tree, the percentage is around 50% in both systems, but for Lee-TM it is between 10%-12%. Since Red-Black tree has invoked the `TM_EXIT` instruction more than Lee-TM, one might expect it to show bigger speedup in Figure 10.1, which is not the case.

Processors	TM-S		TM-U	
	Lee-TM	RB Tree	Lee-TM	RB Tree
2	11.40%	50.11%	12.02%	49.97%
4	11.48%	49.84%	12.00%	49.77%
8	11.33%	50.24%	11.84%	49.73%
16	12.46%	49.58%	10.83%	49.84%

Table 10.2: Usage of `TM_EXIT` as a percentage of total commits

If the size of the write set is small, the amount of time spent committing may not make a significant difference to the overall execution time. Therefore the amount of speculative data committed in both applications is analysed in order to see the effect of removing non-useful commits towards the overall execution time. Table 10.3 shows the number of bytes committed per transaction in both applications in both systems. From Table 10.3 it can be seen that Lee-TM has a significantly bigger write set size in comparison to that of the Red-Black tree. Therefore exiting from a transaction without

committing it, gives a bigger advantage to Lee-TM than it does to Red-Black tree. This makes Lee-TM show better improvements than Red-Black tree.

It can also be seen in Table 10.3 that the number of bytes committed per transaction increases in Lee-TM for both architectures. According to Table 9.1, for Lee-TM, non-useful commits have reduced as the number of cores are increased. This means more transactions have committed successfully, thereby increasing the number of bytes committed per transaction.

Processors	TM-S		TM-U	
	Lee-TM	RB Tree	Lee-TM	RB Tree
2	9519.15	518.38	9742.59	580.00
4	9941.44	529.47	9704.24	588.78
8	11864.72	534.09	9929.95	592.80
16	15961.10	534.14	10148.61	591.32

Table 10.3: Bytes committed per transaction

Not using the interconnect for non-useful commits reduces the contention for it. In other words, one of the overheads incurred by unnecessary commits is the bus contention. In both TM systems the communication arbiter is designed to give the highest precedence to commit requests. Since a commit phase takes time to complete, all the bus requests are denied during this time. This increases the bus contention. Figure 10.2 shows the bus contention presented in both TM systems. In the legend, TM-S-Baseline means that the architecture is TM-S (Section 10.1.2) and it does not support the `TM_EXIT` functionality. Similarly TM-S-TM_EXIT means that the architecture is TM-S (Section 10.1.2) and it supports the `TM_EXIT` functionality. The same applies for TM-U-Baseline and TM-U-TM_EXIT.

It can be seen in Figure 10.2, as expected according to the above discussion, that both applications show less bus contention when `TM_EXIT` functionality is used. This is mainly because of the reduction of commits that are not useful to the completion of the program. Even though both applications show a reduction in bus contention, Lee-TM shows a significant performance improvement in Baseline-2 (TM-U). The reason behind this is, that the transactions in Lee-TM have quite a lot of speculative data that cannot be held in the level 1 cache. Therefore they overflow during the execution of the atomic block. When a transaction becomes longer, more addresses are inserted to the signature. When more addresses are inserted to the signature, it increases the probability of producing false positives. Despite its disadvantage of serializing commits,

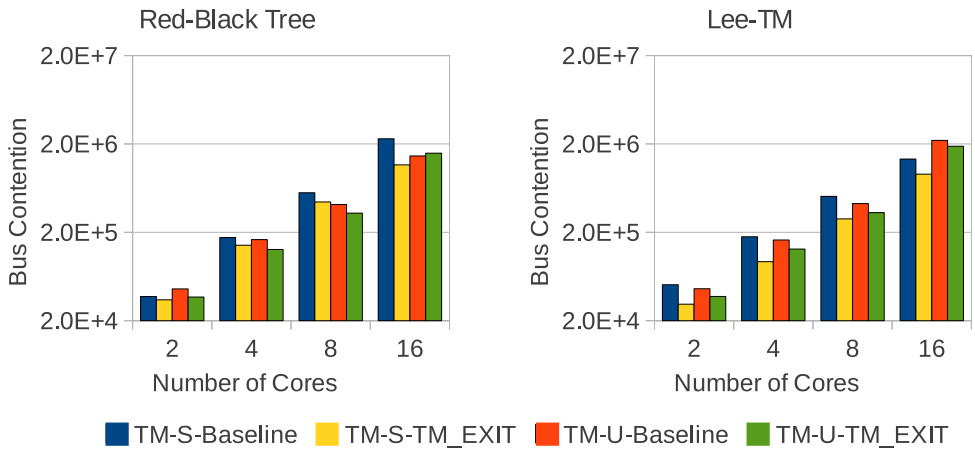

Figure 10.2: Effect on bus contention when using `TM_EXIT`

TM-S has the advantage of having only one large transaction at any given time. This is because, TM-S only allows one transaction to overflow and all others have to wait until this commits. Because of this, the number of false positives are reduced in TM-S. In the case of TM-U, there can be any number of large transactions running at a given time. Therefore this could increase the probability of false positives produced in the system.

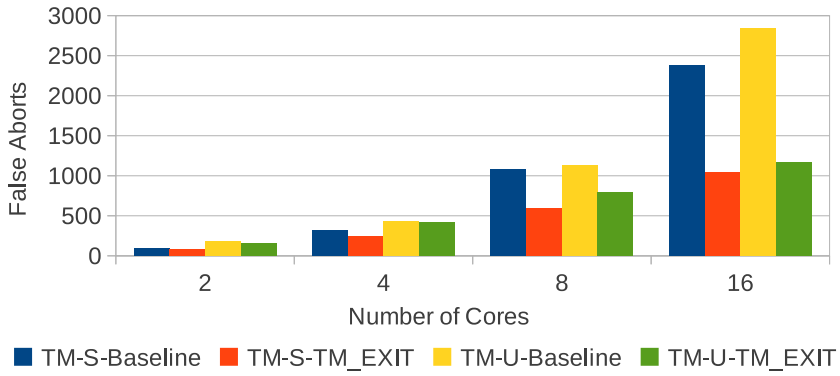

Figure 10.3: Number of false positives occurred in Lee-TM for both baselines and architectures supporting `TM_EXIT`

The number of false positives occurred in Lee-TM for both baselines and for both architectures supporting `TM_EXIT` is shown in Figure 10.3. In the legend, TM-S-Baseline means that the architecture is TM-S (Section 10.1.2) and it does not support the `TM_EXIT` functionality. Similarly TM-S-TM_EXIT means that the architecture is TM-S (Section 10.1.2) and it supports the `TM_EXIT` functionality. The same applies

for TM-U-Baseline and TM-U-TM_EXIT. It can be seen from the figure that, by not invoking a commit phase for non-useful commits, Lee-TM is able to reduce the number of false positives in both architectures which supports the `TM_EXIT` functionality. It can also be seen than TM-U-Baseline has more false abort than the TM-S-Baseline. Even though both `TM_EXIT` architectures are able to reduce the number of false aborts, the reduction is higher in TM-U than that of the TM-S. Therefore the performance improvement of TM-U-TM_EXIT over TM-U-Baseline is higher than that of the TM-S.

10.2.4 Performance Evaluation of Increased Expressiveness

Section 10.2.2 showed the performance improvement when the `TM_EXIT` functionality is integrated to existing TM applications. This section focuses on the effect of using `TM_EXIT` to increase the expressiveness. The Linked-list example, described in Chapter 9, Section 9.5.4, is used for this experiment. The objective of the application is to reverse the linked list, if it has more elements than a threshold value. As shown in Figure 9.12, transactions have been added to the reverse method. If the length of the list is found to be less than the threshold, speculative changes are abandoned and the control is transferred to the end of the atomic block using `TM_EXIT`. For comparison purposes a naive implementation of the code, shown in Figure 9.11, is also executed. As described in Section 10.1.2, the TM-S baseline cannot be used for this experiment. Therefore experiments are carried out using the TM-U baseline. Experiments are made in a system with a single processor.

For this experiment, 200 linked-lists with each having a length less than 10000 are used. Three threshold values are used for the evaluation. In the first configuration the threshold is set to 100, so that more lists needs to be reversed. In the second configuration, the threshold is set to 9000, in order reduce the number of lists that needs to be reversed. In the final configuration, for each list, the threshold is determined randomly. Threshold configurations are summarised in Table 10.4.

Configuration	Threshold
a	100
b	9000
c	random value

Table 10.4: Threshold configurations for Linked-list

Figure 10.4 shows the execution time of the modified linked-list (that uses `TM_EXIT`) normalised to that of the unmodified code. On the X-axis `a`, `b`, `c` stands for the

threshold configurations shown in Table 10.4. From the figure, it can be seen that the modified application outperforms the unmodified one in configurations `a` and `c`. In the case of configuration `b`, the modified application takes more time than the unmodified one.

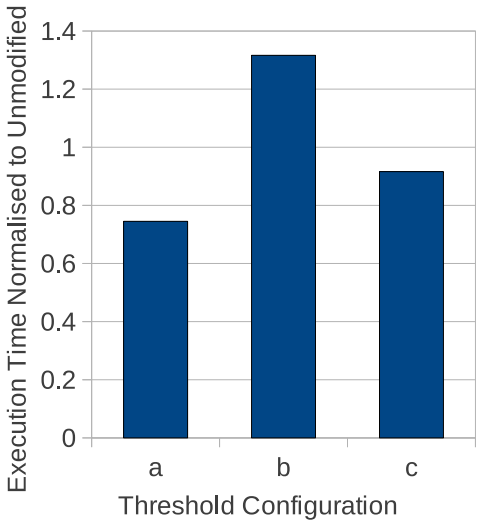

Figure 10.4: Execution time of modified Linked-list normalised to the original

In the case of configuration `a`, since the threshold is a low value, a larger number of lists (196) need to be reversed. Therefore the unmodified application has to traverse 196 lists twice (one to count and one to reverse) whereas the modified application only makes a single traverse for all 200 lists. Therefore in configuration `a`, the application using `TM_EXIT` has an advantage of a reduced number of traverses. In the case of configuration `b`, only 17 lists need to be reversed as the threshold is kept at a higher value. In this case, the unmodified application has only 17 lists to traverse twice. However one could still argue that modified application only does 200 traverses in total whereas, in this case, the unmodified one does 217 traverses in total. Therefore the execution time of the unmodified application should still be higher than the modified one, which is not the case. An explanation for this behaviour is presented later in this section. Finally in configuration `c`, 92 lists were required to be reversed. Since this is less than the requirement in configuration `a`, configuration `c` reports a slightly higher execution time than the former.

The fact that unmodified application has to traverse certain lists two times (one for counting and one for reversing), explains it having a higher execution time than the modified application in configurations `a` and `c`. However this trend has not been presented in configuration `b`. The reason for this behaviour is the increased number of

main memory accesses present in the TM-U architecture.

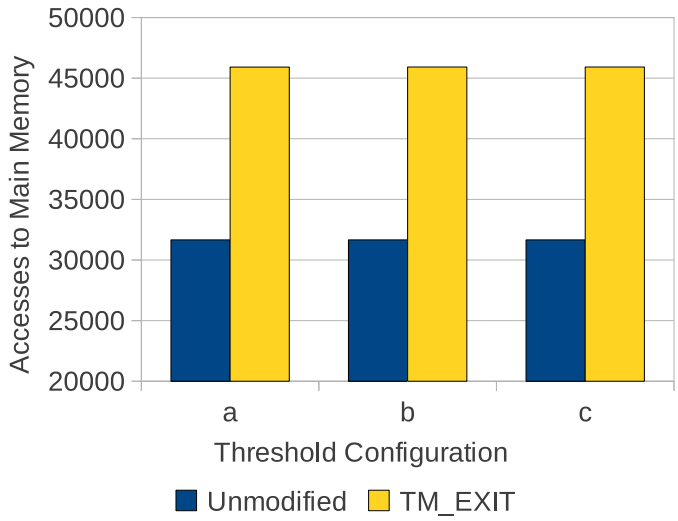

Figure 10.5: Memory accesses of both modified and unmodified Linked-list applications

Figure 10.5 shows total number of main memory accesses reported in both applications. In the legend, Unmodified refers to the memory accesses occurred in the original linked-list and TM_EXIT refers to the memory accesses occurred in the modified linked-list. There, it can be seen that for all the configurations the modified application has a higher number of memory accesses than the unmodified one. In the TM-U baseline, when a transactionally modified entry is evicted before a commit operation happens, it is stored in the uncached area and the *Overflow* bit ('O') in that cache line is set. Thereafter, when a cache miss is occurred for a cache line with the 'O' bit set, it is directed to this overflow area, which is searched for a matching entry. A miss in the uncached area, is treated as a normal cache miss, hence fetched from the original memory location. Since the most of the linked lists used in the experiment are longer, they cannot be held in the L1 cache during the execution of an atomic block. Therefore when a transactional cache overflow happens, the modified cache line is stored in the uncached area of the memory and the 'O' bit in the cache line is set. All the subsequent cache misses whose index bits are similar to this memory location's index go through the uncached area of the memory. Therefore the number of memory accesses present in the modified application increases. This does not apply to the unmodified application as no speculative operations are performed in that case.

Table 10.5 shows the number of times the overflow area is accessed in the modified

Configuration	Overflow Accessed
a	18906
b	18906
c	18907

Table 10.5: Number of times the overflow area is accessed in the Linked-list application that uses TM_EXIT

application. When the overflow area is accessed, a processor is stalled a minimum of 100 cycles, which is the latency of main memory. The situation becomes worse, if the location is not found in the overflow area. In this experiment none of these overflow accesses resulted in a success, which means all of these accesses had to be treated as normal cache misses after receiving the response from the main memory. This adds a performance penalty which is not present for the unmodified application. However this penalty is masked in configurations a and c as the total number of traverses required for the unmodified application is significantly higher than the modified one. Since the same condition does not apply for configuration b, the execution time of the unmodified application is lower than the modified one.

10.3 Summary

When extending to support TM_EXIT, the architectural requirements of two baseline TM systems are discussed in this chapter. Proposed implementation of two TM systems and their applicability on different programming situations is also described. Further it is concluded that one baseline architecture, TM-S cannot be used in certain situations. Even though increasing performance is not a prime objective of the proposal, an experiment made to evaluate the effects of adding TM_EXIT to existing HTM systems is also presented in the chapter. The chapter also characterises the results of these performance evaluations in order to analyse the effect of TM_EXIT on various parameters such as bus contention, false transaction aborts and so on. Finally it also shows the performance effects of using TM_EXIT to improve expressiveness.

Chapter 11

Related Work on TM_EXIT

This chapter summarises the related work on TM_EXIT. Section 11.1 summarises the proposals made in the context of Software Transactional Memory (STM), that are related to the proposed TM_EXIT functionality. Several Hardware Transactional Memory (HTM) approaches and their ability to support TM_EXIT are discussed in Section 11.2. Section 11.3 focuses on the applicability of TM_EXIT to eager versioning HTM systems. Finally Section 11.4 summarises the chapter.

11.1 Software Approaches

Early release [49, 100] has been proposed as a way of reducing contention in both hardware and software TM systems. The objective is that a programmer can remove an entry from the read set of its current transaction. Thereafter any write operation to this location by other transactions will not cause conflicts with the current transaction. If this feature is to be used to provide the same functionality as TM_EXIT, then a programmer has to modify the entire application to remove all the memory locations that are read and written during the current transaction from the read and write sets, if the current transaction is found to be non-useful. First of all this requires a programmer to keep track of all the memory locations that are read and written before the usefulness condition of a transaction. Secondly, this may be possible with STMs and also with some HTMs, but most recent HTMs use signatures to keep track of read and write sets. Signatures are implemented as a fixed width bit representation, in which certain bits are set according to the address being considered. One of the features of signatures is that an element can be added to it, but cannot be removed. Therefore early release cannot be used to provide the same functionality as TM_EXIT.

The TM construct *orElse* [43] is used to execute a second transaction if the first one retries. Therefore by adding a "dummy" transaction as the second one and associating the *usefulness* criterion as the retry condition, a programmer may be able to achieve the objective of `TM_EXIT` with *orElse*. For example in the case of Lee-TM failure to reach the destination in *expand* can be defined as the "retry" condition. In such situations, the dummy transaction would commit, thereby delivering the same performance impact as `TM_EXIT`. However this is achieved at the cost of losing clear and concise code in a program.

Crowl *et al.* proposed to integrate TM semantics in to C++ [26]. There, the authors discuss different ways to exit from a transaction. According to the authors, the "normal" way to end a transaction is to commit it. They also suggest committing a transaction even if it is exited with typical C language keywords like *return, break, continue* and *goto*. In their specification another way to end a transaction is to exit with *longjmp*. The idea is to abandon the speculative operation without finishing it. Thereafter the environment is restored with the one saved by *setjmp*. This is similar to an abort operation requiring the transaction to be restarted. In the case of Lee-TM, if used when the *expand* returns False, this will lead to a live lock. Consider the case where *expand* cannot reach its destination because all the possible paths have been blocked, hence returns false. The same transaction will continue to retry until it succeeds, but as no route exists this will never happen.

TM constructs *__tm_abort* [79] and *user-level aborts* [99] have the same objective as the proposed `TM_EXIT`. The behaviour is, once executed within a critical section, to discard all the speculative modifications and to transfer the control to the statement immediately following the critical section. However in these semantics, the programmer loses the ability to explicitly *abort* and *restart* transactions, they can only abort.

The Xfork [90] framework allows programmers to define logical relationships between sibling transactions. The basic idea is to define nested transactions as AND, OR, or X-OR. When declared as AND all the sibling transactions should be completed in order to commit a transaction, when declared as X-OR only one successful transaction should be committed, and for transactions declared as OR each sibling transaction can fail or succeed independently. If this approach is used in the Lee-TM example, both *expand* and *backtrack* methods can be defined as AND sibling transactions ensuring that the execution of the latter is delayed until the former completes due to data dependencies. The Xfork API for AND guarantees that, if any of the siblings returns false, no transaction will commit and the transactions will retry. Once again in the case

of Lee-TM this can cause a live lock.

Finally, programming language extensions like *abox* [41] have been proposed in order to handle exceptions raised within critical sections. However no direct hardware support is provided for this and `TM_EXIT` fits the required purposes well.

11.2 Hardware Approaches

McDonald *et al.* [73] propose the first Instruction Set Architecture (ISA) for HTM. Along with the functionalities expressed by previously proposed HTMs, the authors suggest three major operations to manage transactions: *xbegin, xcommit, xabort*. As with the *abort_transaction* function in Log-TM [77], the *xabort* instruction executes a code that is registered with the abort handler. The purpose is to allow a programmer to explicitly abort a transaction. If this feature is used to achieve the objective of `TM_EXIT`, a *dummy* function which can explicitly transfer the control to the end of the atomic block needs to be constructed. Such a function would just contain a *goto* statement to move the execution to the end of the block. Thereafter this function needs to be registered with the abort handler. Now, when the *xabort* is invoked, control can be transferred to the end of the atomic block. However, it is still required to notify the hardware not to restart the transaction once the abort function completes because the default behaviour of *xabort* is to do so. Since such a facility is not provided, the proposed ISA cannot support a functionality similar to `TM_EXIT`.

Notary [111] is a TM system which proposes to separate private and shared data and to exclude private data from the read and write sets of a transaction. Their approach relies on using separate virtual pages for shared and private memory locations. If their approach is to be considered, first compiler and/or programming language support is needed to allocate all the private data, including stack, in those private pages. In addition to that, a programmer is required to categorise data into those types. Then for the Lee-TM application, a write set of zero size can be produced, when the *backtrack* phase is not executed. However, excluding local variables from the write set may pose consistency violations in TM. Sanyal *et al.* [96] propose an undo buffer to overcome this problem, but this approach requires extra hardware whose size cannot be determined in advance and also requires modifications to the memory management and to the run time systems.

Hardware support for TM has already been incorporated within Azul chips [25].

Their API also provides an *Abort* instruction which marks all the speculatively modified cache lines as invalid. However, it is not well defined whether the control is transferred to the beginning of the critical section [73] or to the end [99].

Sun's Rock processor [18, 29] also provides TM support. Their design comes with only two extra instructions: *chkpt* and *commit*. When a transaction is started by a *chkpt* instruction, a pc-relative fail address can be registered with the transaction itself. Control is then transferred to this address in case of an abort. They also provide an unconditional trap instruction which provides the facility to cause explicit aborts from software. This fail address feature is not able to provide the functionality of `TM_EXIT` because it has to be registered at the beginning of the transaction. A programmer cannot determine at that time whether a particular transaction is going to be useful or not. Certain modifications are required to extend this fail-address feature to provide the `TM_EXIT` functionality. The first modification is to register two pc-relative fail addresses one pointing to the beginning of the transaction and the other pointing to the end of it. Later when an explicit abort is invoked from the use code, an indication needs to be made stating whether to retry or to exit the transaction. Then depending on this indication, the abort mechanism will decide which operation to perform.

The Advanced Synchronization Facility(ASF) [1] is a proposal for extending hardware support for lock free data structures. They introduce 7 new instructions including *ABORT*. Like with the ISA proposed by McDonald *et al.* [73], the *ABORT* instruction rolls back the speculative region and the state is restored using the snapshot taken when the *SPECULATE* instruction was executed. The control is then transferred to the instruction preceding the *SPECULATE* instruction. This instruction is a *JNZ* instruction which will jump to a handler, as the *ABORT* has set the zero flag. This handler can then decide based on the flags set by the *ABORT* what it should do next. Jumping to the end of the transaction is an option, but there are no flags to indicate this in the current version of ASF. As a result, while the required changes are small, currently ASF cannot support a functionality similar to `TM_EXIT`.

The *XABORT* instruction in Intel's Haswell [55] allows a programmer to explicitly abort the transaction from the user code. The cause of the abort is communicated to the software using the EAX register. When an abort happens, the execution is resumed at the fallback address registered when the *XBEGIN* instruction is executed. This is conceptually similar to the fail-address of the Rock processor. Modifications required for Haswell in order to support the objective of `TM_EXIT`, are similar to those required for ASF and Rock.

IBM recently presented speculation support in their Blue Gene/Q chip [80]. Neither the ISA additions nor the API is available to discuss how to achieve the objective of TM_EXIT in the proposed system.

11.3 Applicability of TM_EXIT on other TM Systems

The motivation and evaluation of TM_EXIT only focused on HTM systems that support lazy versioning. This section considers the applicability of the TM_EXIT proposal to eager versioning HTM systems. In an eager system all modifications are made in-place, thereby reducing the commit overhead. In such a system when TM_EXIT is invoked it has to discard all the speculatively written entries as in a lazy versioning system. If the transaction fits in the L1 cache the cost of this process is the same for both eager versioning and lazy versioning HTMs. If a transaction has overflowed the cache, for eager versioning HTMs this involves reading the original value from a separate log and replacing the modified entry with this value. For lazy versioning HTMs this depends on how the overflows are handled. For example in the TM-S baseline it is not possible to restore such memory locations as the original value is not recorded. In the case of the TM-U baseline, this involves only clearing the overflow area of the memory. This means that for lazy versioning HTMs like TM-U, there is an advantage over the eager ones when a transaction does not fit in the L1 cache. However, this costly step is only required if the operations performed outside the usefulness criteria are accessible by other threads. This is not the case for benchmarks like Lee-TM and Red-Black tree. However for applications like the Linked-list example, restoration of overflowed memory locations is required for correctness purposes.

Avoiding the commit phase for non-useful transactions reduces the bus contention which counts for a certain fraction of the reported speedups. In the case of eager versioning HTMs this will not result in a direct advantage as it does for lazy ones. However even eager versioning HTMs will get some benefit if the TM_EXIT functionality is incorporated. For example, consider a situation where a transaction fits in the L1 cache of an eager HTM system. Even though the transaction is not useful, a commit operation is performed. Since the transaction fits in the cache, no communication is done at the commit phase. For simplicity, assume the cache is filled with transactionally modified entries. Later when a cache miss is encountered space has to be allocated in the current cache by writing back an existing entry. Even though this entry is modified, the value has no use as it belongs to a non-useful transaction. If TM_EXIT

functionality is provided, it could have cleared all these entries thereby avoiding this communication. A similar situation where `TM_EXIT` can be useful to eager HTMs is when a context switch happens after the commit phase of a non-useful transaction. In such situations all the dirty cache entries need to be saved before allocating space for cache requests of the new process. This saving of state requires communication, if `TM_EXIT` functionality is provided this communication can be avoided by clearing the dirty cache entries of non-useful transactions.

11.4 Summary

From the survey presented in this chapter, it can be seen that there is no direct hardware proposal to avoid committing non-useful transactions. Even though a proposal for such a semantic has been proposed at the programming language level [79, 99], it has not been incorporated within hardware TM proposals. It is worth noting that such a feature is more valuable in HTM than in STM, because STM systems can produce a write set of zero size for Lee-TM when the *backtrack* phase is not executed. This is because when using STM, the application code is instrumented in a way that shared data is accessed using special read/write operations as in ASF [1]. In the case of Lee-TM all the operations on shared data is performed within the *backtrack* method, hence STMs are able to produce a write set of zero size when it is not executed. This is not the case with most of the hardware TM systems.

Chapter 12

Conclusions and Future Work

The major contributions of this thesis are centred around Hardware Transactional Memory (HTM). Contributions are grouped into three categories which are orthogonal. These three categories were presented as three separate parts in the thesis. Part I presented the concept, design and evaluation of Data Centric Transactional Memory (DaCTM). The conclusions and the possible future research directions of DaCTM are presented in Section 12.1. Part II of the thesis presented the concept, design and evaluation of SnCTM, a novel way of reducing false aborts in hardware signatures. Section 12.2 presents the conclusions and future research directions of SnCTM. Part III of the thesis makes a case for having a functionality (`TM_EXIT`) to exit form a transaction without committing it, in the context of HTM. The conclusions and the possible future research directions of this functionality are discussed in Section 12.3.

12.1 Data Centric Transactional Memory

Part I of the thesis presented the concept and the design of DaCTM, a novel architecture that associates the required levels of coherence, consistency and synchronization of each memory location with its access pattern. The idea is to group sets of memory locations having similar characteristics and to allocate them in a suitable region of memory so that the underlying hardware can select different operations based on the region of each location. Even though the current version supports only four memory regions (LO, RO, WNRL, CRW), if a need arises, more regions can be introduced. In DaCTM the data centric approach is coupled with transactional memory.

The thesis evaluated how an architecture can benefit if this region information is communicated to the hardware. As the compiler support is yet to be developed, for

the evaluation, associating a *type* with each memory location was done manually. The output produced by the benchmarks with *type* information associated, is the same as the one produced by unmodified benchmarks. The evaluation showed that DaCTM scaled better and delivered better speed-ups (upto 4.52X) over an optimized lazy-lazy TM system (see Figure 5.2). The thesis experimentally validated the hypothesis of associating the access pattern of a memory location with its required level of coherence, consistency and synchronization to derive a next generation computing model.

The following contributions were made in the Part I of the thesis:

- A mechanism to maintain coherence and consistency based on memory regions is introduced. In this approach the address space of a program can be viewed as a collection of non-overlapping memory regions, each having a predefined level of coherence and consistency. The union of all the regions is equal to the available address space.

- An application programming interface (API) to manage the memory regions, is also introduced. In this way the programmer is relieved from manually managing different memory spaces.

- As the third contribution, a proposal is made to attach scratch-pad memories (SPMs) to each processor to implement one type of memory region (LO). This removes the need to use the interconnect for memory accesses related to this region, thereby reducing the contention.

- Overall design of the architecture to support the above mentioned region-based coherence and consistency, is presented as the last contribution. The evaluation of DaCTM shows that with the proposed approach, bus utilization and contention, processor idle time and false transaction aborts can be greatly reduced thereby aiding scalability. The performance evaluation presents improvements of up to 4.52 times speed-up over an optimized baseline TM system that uses lazy versioning and lazy conflict detection (an improved TCC [39]).

Most of the applications used for the evaluation, scaled upto 16 processors (see Figure 5.1). Also it outperformed both CS and U versions of the baseline, even if the latter had a bigger cache (see Figure 5.2). The hypothesis of DaCTM is based on the fact that the access pattern of a memory location represents the required level of Synchronization, Coherence and Consistency for that location, hence maintaining a global view of the whole shared memory is not required. Figure 5.3 shows the experimental

validation of this. Finally, the speedups of DaCTM over the baseline comes collectively from reduced processor idle time (see Figure 5.4), reduced bus usage (see Figure 5.6) and contention (see Figure 5.5) and reduction of signature insertions (see Figures 5.8 and 5.9) which ultimately resulted in producing less false positives (see Figures 5.10 and 5.11).

The whole concept of DaCTM goes from the high level programming language to the low level system architecture. Even though the concept of DaCTM is discussed using programming language examples, the thesis only considered the architectural aspects of it. The experiments has shown that DaCTM has the potential to solve the scalability issues imposed in multi-core processors when maintaining a global view of the shared memory using conventional cache coherence protocols. Benchmarks had to be manually modified to be able to work with DaCTM architecture. Therefore implementing programming language extensions to support different types of memories or developing escape analysis techniques to extract this information at the compile time are strong candidates for possible future directions of DaCTM.

In the current version of DaCTM, once a type (LO, RO, WNRL, CRW) is associated with a memory location it remains fixed throughout the lifetime of that memory location. Therefore if a memory location is accessed several times, it will be declared as CRW even if only one of these accesses are concurrent. Therefore a future version of DaCTM should have the capability to change the type of a memory location according to the changing access pattern of it. This way coherence and consistency can be maintained for each memory location only when the location requires it and to the degree which it is required.

12.2 Adaptive Sources for Conflict Detection

Part II of the thesis presented the concept and the design of SnCTM, a novel way of reducing false aborts by adaptively changing the source used during the conflict detection stage. The idea is to decide at the time of committing which source to use, *i.e.* cache line or signature. This way the use of signatures is limited to situations where speculative data cannot be held in the local cache.

Part II of this thesis made following contributions:

- The concept of adaptively changing the source of information used to detect conflicts in a hardware TM system, is introduced. It also shows how an existing TM architecture can be extended to support the SnCTM concept.

- The performance evaluation of SnCTM shows improvements of up to 4.62 and 2.93 times speed-up over a baseline TM using lazy versioning and lazy conflict detection (an improved TCC [39]) with two commonly used signature configurations.
- SnCTM gives the opportunity to reduce the size of a signature without compromising the performance. A sensitivity analysis shows that SnCTM with a 64 bit signature can deliver performance comparable to a perfect signature of 8k bits.

The SnCTM concept is evaluated using the STAMP benchmark suite and Lee-TM. The evaluation showed that the SnCTM proposal delivers better speed-ups (up to 4.62 and 2.93) over an optimized lazy-lazy TM system with two commonly used signature configurations (see Figure 8.4). The number of transactional aborts is reduced by using the SnCTM approach, by means of reducing false aborts (see Figures 8.5 and 8.6). For some applications, the number of false positives occurring in SnCTM is similar to a system with a perfect (8k) signature (see Figure 8.7).

In addition to measuring the effect of SnCTM on the execution time, a sensitivity analysis of the signature length is performed. The results of both baseline and SnCTM is compared against a perfect signature. There it shows that even with a smaller signature (64 bit) SnCTM was able to deliver performance comparable to a perfect system whereas the baseline suffers huge performance degradation (see Figures 8.8 and 8.9). A further investigation shows that SnCTM achieves this by reduction of processor idle time (see Figure 8.10). Another important aspect of the SnCTM proposal is that it is independent of the underlying signature implementation. Therefore all the proposed techniques [19, 20, 61, 84, 85, 86, 111] to improve the efficiency of a signatures can be used in SnCTM as well.

SnCTM uses the H3 [12] class of hash functions as its signature implementation. Several approaches can be found in the HTM literature proposing different signature implementations that can reduce the number of false positives. It would be an interesting experiment to see how much benefit can be gained from integrating those approaches with the signature implementation of SnCTM. Further, SnCTM requires additional area for having both read and write sets and signatures. It also requires some control logic to decide which source to use for detecting conflicts. Another possible experiment is to conduct a cost (area)-benefit (performance) analysis of the SnCTM approach. The same experiment can later be extended by including a third parameter (signature length) to the equation.

12.3 Exiting a Transaction without Committing

Part III has presented a case for the TM_EXIT function to be added to hardware TM systems. The objective of the functionality is to exit from a transaction without committing it. Once invoked within a transaction, all the speculative changes are abandoned and the control is transferred to the line immediately following the atomic block. Unnecessary commits can be avoided in this manner, thereby resulting in less interconnect usage. In addition to making the case, it also discussed the feasibility of integrating this functionality with two HTM systems.

The following contributions were made in Part III of the thesis.

- A successful case is made for supporting TM_EXIT functionality in HTM.
- In addition to extending the existing code segments to use TM_EXIT, a case has been presented where the expressiveness can be increased using the proposed functionality.
- As the third contribution, the feasibility of integrating TM_EXIT with two baseline HTM systems is presented and the proposed implementation is discussed.
- As the final contribution, performance evaluations of TM_EXIT on two HTMs are presented.

The performance impact of TM_EXIT is measured using Lee-TM and a transactional version of the Red-Black tree with two hardware TM systems. The results showed that with hardware support for TM_EXIT, a speedup of up to 2.28X can be achieved for the applications tested (see Figure 10.1). This speedup is gained from a combination of lower false positives (see Figure 10.3) and lower bus contention (see Figure 10.2) which ultimately results in less wasted time.

The effect of increased expressiveness is also measured with a contrived example using a linked-list. Depending on the input configuration of the linked-list, the speedup varied from 0.75X to 1.31X (see Figure 10.4). The reduction of the speedup is due to the HTM (similar to LTM [3]) having to access the uncached area of memory when the cache line has the 'overflow' bit set (see Figure 10.5 and Table 10.5).

It would be an interesting study to analyse the available TM benchmarks to identify places where TM_EXIT can be useful. Similarly, analysing some parallel benchmarks (TM and non-TM) with the intention of increasing the expressiveness using TM_EXIT would also be an interesting topic. The evaluation presented in the thesis used two variations of a lazy-lazy HTM system. Therefore implementing and evaluating TM_EXIT

in other types of research (eg: LogTM [77]) and industrial (AMD-ASF [21], Intel Haswell [55]) HTMs, is also a strong candidate for future research directions of `TM_EXIT`.

Bibliography

[1] Advanced Micro Devices. AMD Advanced Synchronization Facility Proposal. `http://developer.amd.com/tools/ASF/Pages/default.aspx`, 2009.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29:66–76, December 1996.

[3] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. *IEEE Micro*, 26:59–69, 2006.

[4] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. SharC: Checking Data Sharing Strategies for Multithreaded C. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 149–158, New York, NY, USA, 2008. ACM.

[5] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: Design Alternative for Cache on-chip Memory in Embedded Systems. In *CODES '02: Proceedings of the 10th International Symposium on Hardware/Software Codesign*, pages 73–78, New York, NY, USA, 2002. ACM.

[6] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a Scalable Memory Allocator for Multithreaded Applications. *SIGPLAN Not.*, 35:117–128, November 2000.

[7] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[8] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-Consistent Distributed Shared Memory. In *Proceedings of the 10th International Parallel Processing Symposium*, IPPS '96, pages 132–141, Washington, DC, USA, 1996. IEEE Computer Society.

[9] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 24–34, New York, NY, USA, 2007. ACM.

[10] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 127–138, Washington, DC, USA, 2008. IEEE Computer Society.

[11] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 246–257, Washington, DC, USA, 2005. IEEE Computer Society.

[12] J. Lawrence Carter and Mark N. Wegman. Universal Classes of Hash Functions (Extended Abstract). In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, STOC '77, pages 106–112, New York, NY, USA, 1977. ACM.

[13] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. Colorama: Architectural Support for Data-Centric Synchronization. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, pages 133–144, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 278–289, New York, NY, USA, 2007. ACM.

[15] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Luis Ceze, Christoph von Praun, Călin Caşcaval, Pablo Montesinos, and Josep Torrellas. Concurrency Control with Data Coloring. In *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: held in conjunction with the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, MSPC '08, pages 6–10, New York, NY, USA, 2008. ACM.

[17] 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 *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, HPCA '07, pages 97–108, Washington, DC, USA, 2007. IEEE Computer Society.

[18] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT Processor. *Micro, IEEE*, 29(2):6 –16, March-April 2009.

[19] Woojin Choi and Jeff Draper. Locality-aware Adaptive Grain Signatures for Transactional Memories. In *Proceedings of the 24th International Parallel and Distributed Processing Symposium*, pages 1–10, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[20] Woojin Choi and Jeff Draper. Unified Signatures for Improving Performance in Transactional Memory. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 817–827, Washington, DC, USA, 2011. IEEE Computer Society.

[21] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 27–40, New York, NY, USA, 2010. ACM.

[22] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded Page-based Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and*

Operating Systems, ASPLOS-XII, pages 347–358, New York, NY, USA, 2006. ACM.

[23] Jaewoong Chung, David Christie, Martin Pohlack, Stephan Diestelhorst, Michael Hohmuth, and Luke Yen. Compilation of Thoughts about AMD Advanced Synchronization Facility and First-Generation Hardware Transactional Memory Support. In *TRANSACT '10: 5th ACM SIGPLAN Workshop on Transactional Computing*, April 2010.

[24] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 39–50, Washington, DC, USA, 2010. IEEE Computer Society.

[25] Cliff Click. HTM Will Not Save the World. *Presentation at TMW10 Workshop*, May 2010.

[26] Lawrence Crowl, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Integrating Transactional Memory into C++. In *TRANSACT '07: 2nd ACM SIGPLAN Workshop on Transactional Computing*, August 2007.

[27] Blas A. Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José F. Duato. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 93–104, New York, NY, USA, 2011. ACM.

[28] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[29] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceeding of the 14th International Conference on Architectural Support for Programming*

Languages and Operating Systems, ASPLOS '09, pages 157–168, New York, NY, USA, 2009. ACM.

[30] Laura Effinger-Dean and Dan Grossman. Region-Based Dynamic Separation for STM Haskell. In *TRANSACT '11: 6th ACM SIGPLAN Workshop on Transactional Computing*, June 2011.

[31] Magnus Ekman, Per Stenström, and Fredrik Dahlgren. TLB and Snoop Energy-Reduction using Virtual Caches in Low-Power Chip-Multiprocessors. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, ISLPED '02, pages 243–246, New York, NY, USA, 2002. ACM.

[32] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[33] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 347–358, New York, NY, USA, 2010. ACM.

[34] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, 1991.

[35] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *SIGARCH Comput. Archit. News*, 18(3a):15–26, 1990.

[36] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 162–171, Washington, DC, USA, 1999. IEEE Computer Society.

[37] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the 7th International Conference on Very Large Data Bases - Volume 7*, VLDB '1981, pages 144–154. VLDB Endowment, 1981.

[38] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with Transactional Coherence and Consistency (TCC). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XI, pages 1–13, New York, NY, USA, 2004. ACM.

[39] 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*, ISCA '04, pages 102–113, Washington, DC, USA, 2004. IEEE Computer Society.

[40] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 184–195, New York, NY, USA, 2009. ACM.

[41] Derin Harmanci, Vincent Gramoli, and Pascal Felber. Atomic Boxes: Coordinated Exception Handling with Transactional Memory. In *Proceedings of the 25th European Conference on Object Oriented Programming*, ECOOP'11, pages 634–657, Berlin, Heidelberg, 2011. Springer-Verlag.

[42] Tim Harris, James Larus, and Ravi Rajwar. Transactional Memory, 2nd Edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[43] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

[44] Isuru Herath, Demian Rosas-Ham, Daniel Goodman, Mikel Luján, and Ian Watson. A case for Exiting a Transaction in the Context of Hardware Transactional

Memory. In *TRANSACT '12: 7th ACM SIGPLAN Workshop on Transactional Computing*, February 2012.

[45] Isuru Herath, Demian Rosas-Ham, Mikel Luján, and Ian Watson. SnCTM: Reducing False Transaction Aborts by Adaptively Changing the Source of Conflict Detection. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 65–74, New York, NY, USA, 2012. ACM.

[46] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[47] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

[48] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–, Washington, DC, USA, 2003. IEEE Computer Society.

[49] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[50] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[51] Mark D. Hill. Multiprocessors Should Support Simple Memory-Consistency Models. *Computer*, 31:28–34, August 1998.

[52] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008.

[53] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. McRT-Malloc: a Scalable Transactional Memory Allocator. In *ISMM '06: Proceedings of the 5th International Symposium on Memory Management*, pages 74–83, New York, NY, USA, 2006. ACM.

[54] The IEEE and The Open Group. The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition. `http://www.opengroup.org/onlinepubs/000095399/basedefs/pthread.h.html`, 2001-2004.

[55] Intel Corporation. Intel Architecture Instruction Set Extensions Programming Reference. `http://software.intel.com/file/41417`, 2012.

[56] Daehoon Kim, Jeongseob Ahn, Jaehong Kim, and Jaehyuk Huh. Subspace Snooping: Filtering Snoops with Operating System Support. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 111–122, New York, NY, USA, 2010. ACM.

[57] Tom Knight. An Architecture for Mostly Functional Languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM.

[58] Rakesh Kumar, Timothy G. Mattson, Gilles Pokam, and Rob Van Der Wijngaart. The Case for Message Passing on Many-Core Chips. Technical Report UILU-ENG-10-2203 (CRHC-10-01), University of Illinois Urbana-Champaign, 2010.

[59] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid Transactional Memory. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[60] Bradley C. Kuszmaul and Charles E. Leiserson. Transactions Everywhere. Technical report, MIT Laboratory for Computer Science, 2003. `http://hdl.handle.net/1721.1/3692`.

[61] Martin Labrecque, Mark C. Jeffrey, and J. Gregory Steffan. Application-Specific Signatures for Transactional Memory in Soft Processors. In *6th International Symposium on Applied Reconfigurable Computing (ARC'10)*, pages 42–54, 2010.

[62] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[63] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. *See* `http://llvm.cs.uiuc.edu`.

[64] Chris Lattner and Vikram Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 129–142, New York, NY, USA, 2005. ACM.

[65] Kyungwoo Lee, Xing Fang, and Samuel P. Midkiff. Practical Escape Analyses: How Good are They? In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 180–190, New York, NY, USA, 2007. ACM.

[66] Kyungwoo Lee and Samuel P. Midkiff. A Two-Phase Escape Analysis for Parallel Java Programs. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 53–62, New York, NY, USA, 2006. ACM.

[67] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased Transactional Memory. In *TRANSACT '07: 2nd ACM SIGPLAN Workshop on Transactional Computing*, August 2007.

[68] David J. Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Comput. Surv.*, 25:303–338, September 1993.

[69] David B. Lomet. Process Structuring, Synchronization, and Recovery using Atomic Actions. In *Proceedings of an ACM conference on Language Design for Reliable Software*, pages 128–137, New York, NY, USA, 1977. ACM.

[70] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, February 2002.

[71] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 182–193, New York, NY, USA, 2003. ACM.

[72] Alex Matveev, Ori Shalev, and Nir Shavit. Dynamic Identification of Transactional Memory Locations. Unpublished Manuscript, Tel-Aviv University, 2007.

[73] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 53–65, Washington, DC, USA, 2006. IEEE Computer Society.

[74] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC 2008, pages 35–46, September 2008.

[75] 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*, ISCA '07, pages 69–80, New York, NY, USA, 2007. ACM.

[76] Gordon E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82 –85, jan. 1998.

[77] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, HPCA, pages 254 – 265. IEEE Computer Society, February 2006.

[78] Andreas Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 234–245, Washington, DC, USA, 2005. IEEE Computer Society.

[79] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 195–212, New York, NY, USA, 2008. ACM.

[80] Martin Ohmacht. Hardware Support for Transactional Memory and Thread-Level Speculation in the IBM Blue Gene/Q System. *Presentation at Wild and Sane Ideas in Speculation and Transactions Workshop*, October 2011.

[81] Kunle Olukotun and Lance Hammond. The Future of Microprocessors. *Queue*, 3(7):26–29, September 2005.

[82] OpenMP Architecture Review Board. OpenMP.org. `http://openmp.org/wp/`, 2009.

[83] Oracle. Java SE Technical Documentation. `http://download.oracle.com/javase/`, 2011.

[84] Lois Orosa, Elisardo Antelo, and Javier D. Bruguera. FlexSig: Implementing Flexible Hardware Signatures. *ACM Trans. Archit. Code Optim.*, 8(4):30:1–30:20, January 2012.

[85] Ricardo Quislant, Eladio Gutierrez, Oscar Plata, and Emilio L. Zapata. Improving Signatures by Locality Exploitation for Transactional Memory. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 303–312, Washington, DC, USA, 2009. IEEE Computer Society.

[86] Ricardo Quislant, Eladio Gutierrez, Oscar Plata, and Emilio L. Zapata. Multiset Signatures for Transactional Memory. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 43–52, New York, NY, USA, 2011. ACM.

[87] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[88] Ravi Rajwar and James R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-X, pages 5–17, New York, NY, USA, 2002. ACM.

[89] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[90] Hany Ramadan and Emmett Witchel. The Xfork in the Road to Coordinated Sibling Transactions. In *TRANSACT '09: 4th ACM SIGPLAN Workshop on Transactional Computing*, February 2009.

[91] Torvald Riegel, Christof Fetzer, and Pascal Felber. Automatic Data Partitioning in Software Transactional Memories. In *SPAA '08: Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, pages 152–159, New York, NY, USA, 2008. ACM.

[92] Peter Rundberg and Per Stenström. Speculative Lock Reordering: Optimistic Out-of-Order Execution of Critical Sections. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 11.1–, Washington, DC, USA, 2003. IEEE Computer Society.

[93] Yair Sade, Mooly Sagiv, and Ran Shaham. Optimizing C Multithreaded Memory Management Using Thread-Local Storage. In *Proceedings of the 14th International Conference on Compiler Construction*, CC'05, pages 137–155, Berlin, Heidelberg, 2005. Springer-Verlag.

[94] Alexandru Salcianu and Martin Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 12–23, New York, NY, USA, 2001. ACM.

[95] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing Signatures for Transactional Memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 123–133, Washington, DC, USA, 2007. IEEE Computer Society.

[96] Sutirtha Sanyal, Sourav Roy, Adrian Cristal, Osman S. Unsal, and Mateo Valero. Dynamically Filtering Thread-Local Variables in Lazy-Lazy Hardware Transactional Memory. In *HPCC '09: Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, pages 171–179, Washington, DC, USA, 2009. IEEE Computer Society.

[97] William N. Scherer, III and Michael L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[98] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[99] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. Towards Transactional Memory Semantics for C++. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 49–58, New York, NY, USA, 2009. ACM.

[100] Travis Skare and Christos Kozyrakis. Early Release: Friend or Foe? In *TRANSACT '06: 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[101] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[102] Bjarne Steensgaard. Thread-Specific Heaps for Multi-Threaded Programs. In *Proceedings of the 2nd International Symposium on Memory Management*, ISMM '00, pages 18–24, New York, NY, USA, 2000. ACM.

[103] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, 23:12–24, June 1990.

[104] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel Distrib. Technol.*, 1(4):58–71, November 1993.

[105] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. EazyHTM: Eager-Lazy Hardware Transactional Memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 145–155, New York, NY, USA, 2009. ACM.

[106] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating Synchronization Constraints with Data in an Object Oriented Language. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 334–345, New York, NY, USA, 2006. ACM.

[107] David W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IV, pages 176–188, New York, NY, USA, 1991. ACM.

[108] Ian Watson, Chris Kirkham, and Mikel Luján. A Study of a Transactional Parallel Routing Algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 388–398, Washington, DC, USA, 2007. IEEE Computer Society.

[109] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[110] 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 *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 261–272, Washington, DC, USA, 2007. IEEE Computer Society.

[111] Luke Yen, Stark C. Draper, and Mark D. Hill. Notary: Hardware Techniques to Enhance Signatures. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 234–245, Washington, DC, USA, 2008. IEEE Computer Society.

[112] Jason Zebchuk, Elham Safi, and Andreas Moshovos. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 314–327, Washington, DC, USA, 2007. IEEE Computer Society.